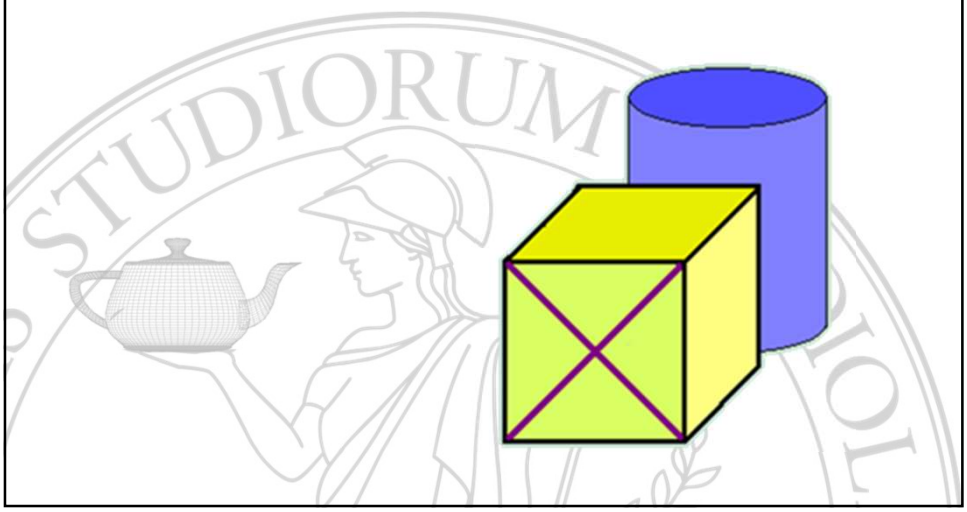
Marco Tarini - Computer Graphics 2024/2025
Università degli Studi di Milano

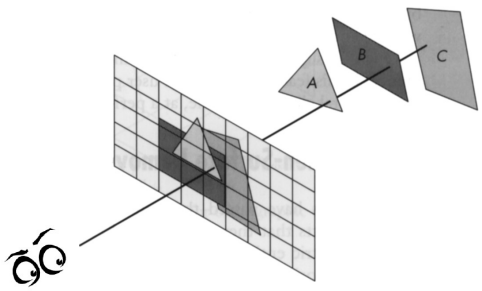
Rimozione Superfici Nascoste



1

Rimozione delle superfici nascoste (cioè occluse)

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



Con raytracing era ovvio ottenere questo effetto!
Con raserization?

2

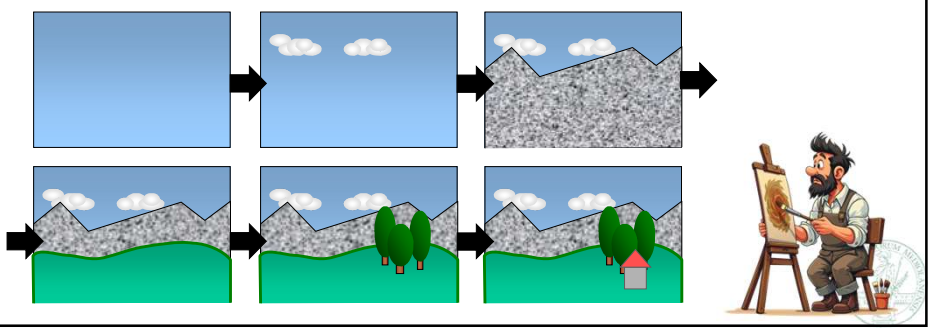
Rimozione delle superfici nascoste ("hidden surface removal")

- ✓ Negli approcci basati su ray-tracing, questo è banale da ottenere:
  - ⇒ basta prendere, per ogni raggio primario, la primitiva intersecata avente la distanza *k* minore
- ✓ Negli approcci basati sulla rasterizzazione, non è così banale
  - ⇒ Se i pixel prodotti dalla rasterizzazione di una primitiva **sovrascrivono** sullo screen buffer quelli generati (alla stessa posizione) dalle primitive precedenti…
  - ⇒ …allora il valore finale sullo screen buffer sarà semplicemente il valore dei pixel generati dall'ultima primitiva rasterizzata
  - ⇒ …e perché questo risultato sia quello corretto, è necessario rasterizzare per ultime le primitive più vicine

3

Rimozioni delle superfici nascoste

- ✓ Prima classe di soluzioni: basate su ordinamento
  - ⇒ le primitive rasterizzate **sovrascrivono** nel frame buffer quelle rasterizzate in precedenza
  - ⇒ quindi, basta rasterizzare le primitive nell'*ordine giusto*
    - ordine detto "back-to-front"
  - ⇒ approccio noto come "**algoritmo del pittore**"



5

Algoritmo del pittore (o depth sorting)

✓ Data una scena (composta da primitive)

1. ordinare le primitive back-to-front
   - dalle più lontane dalla camera, alle più vicine

⇒quindi, in ordine di coordinata Z…

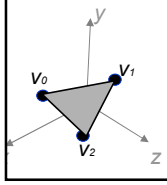
- *crescente*, se in **spazio vista**
  dato che la Z del nostro spazio vista decresce col crescere della distanza dalla camera
- *decrescente*, in **spazio clip** o **spazio schermo**
  dato che, la Z cresce col decrescere della distanza dalla camera
  (in spazio schermo: è detta «depth» e va da 0 a 1;
  in spazio clip: va da –1 a +1)

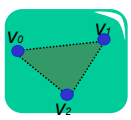
2. rasterizzarle in successione
   - Permettendo a tutti frammenti generati di sovrascrivere i pixel già presenti sul buffer (prodotti dai frammenti generati in precedenza)

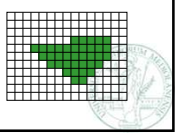
6

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.



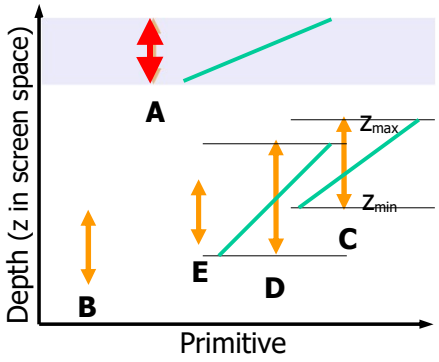




7

Algoritmo del pittore (depth sorting)

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

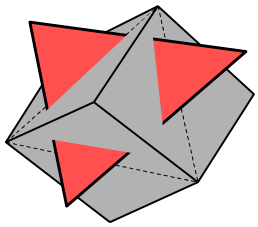


8

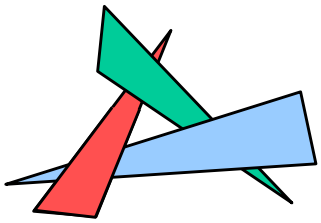
Algoritmo del pittore (depth sorting)

- ✓ Esiste sempre un ordine corretto?

⇒NO.
Controesempio: intersezioni



⇒NO.
Controesempio: cicli



9

Rimozioni delle superfici nascoste tramite ordinamento su Z: sommario degli ostacoli

- ✓ Limiti delle soluzioni basate su oridinamento:
  - ⇒ Problema 1: ordinamento costa $O(n \log n)$ (è "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 trasformazione
  - ⇒ 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 dà i risultati corretti
  - ⇒ Problema 4: complica il rendering
    - prescrivendo un certo ordine di rendering delle primitive

10

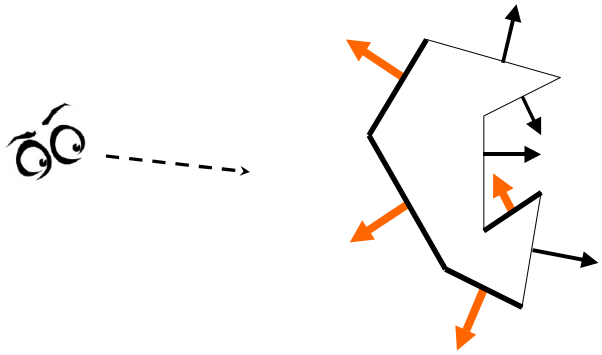
Algoritmo del pittore (depth sorting)

- ✓ 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: mesh ottenuta poligonalizzando un campo di altezza
- ✓ Vediamo invece alternative per rimuovere le superfici nascoste che sono *«order independent»*
  - ⇒ Il rendering produce lo stesso risultato, *indipendentemente* dall'ordine delle primitive

11

Caso particolare per mesh chiuse e ben orientate. Backface Culling

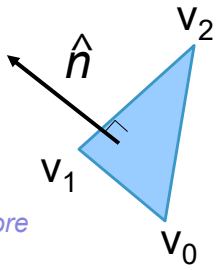
- ✓ E' un caso particolare di rimozione delle superfici nascoste, che abbiamo già visto
- ✓ Idea: se la superficie **chiusa** e **opaca**…
  (e la camera è all'esterno)
  allora, non ne vedrò mai l'interno!



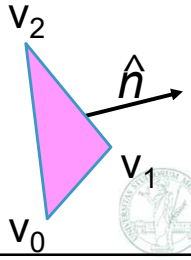
12

Back-face culling

- ✓ Traingolo *front-facing*
  - ⇒"visto da davanti"



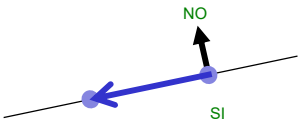
- ✓ Traingolo *back-facing*
  - ⇒"visto di spalle"

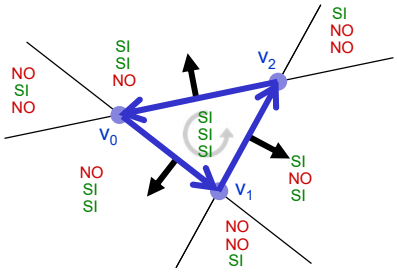


16

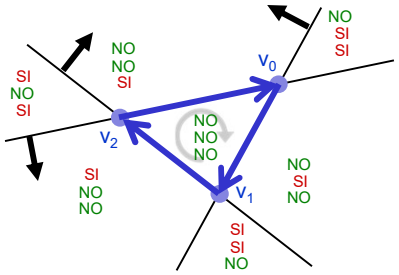
Back-face culling e test di appartenenza di frammento a triangolo

- ✓ 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)

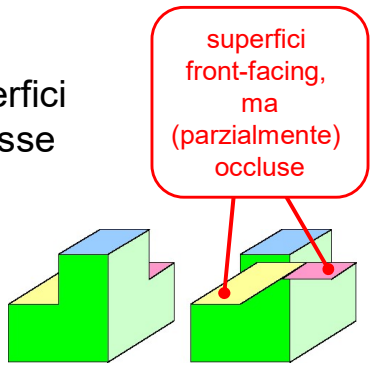


Triangolo 2D back facing
(in spazio Clip)

18

Il back-face culling non è sufficiente (a rimuovere tutte le superfici nascoste)

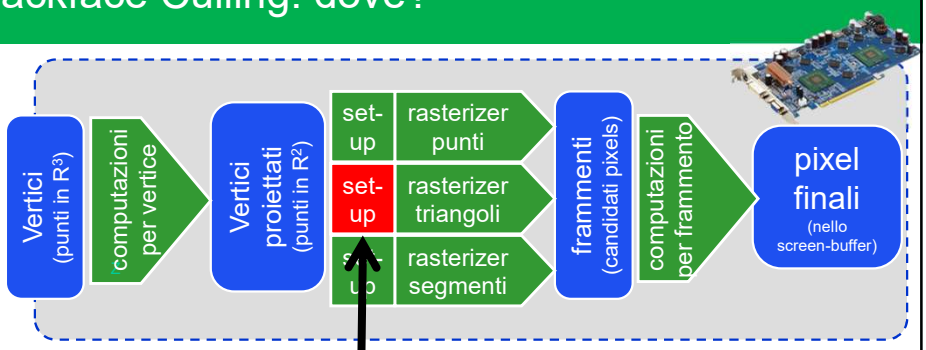
- ✓ Contro-esempio:
  - ⇒succede solo con le superfici concave, cioè non convesse



- ✓ Tuttavia, è utile come ottimizzazione:
  - ⇒mi consente di non processare alcune primitive, perché so che verranno comunque occluse

20

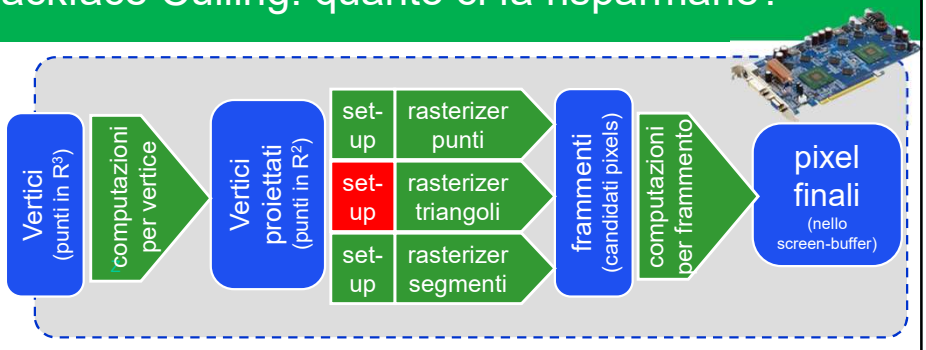
Backface Culling: dove?



QUI.
(in una fase di set-up che precede la rasterizzazione di ciascun triangolo)
(nota: non è generalizzabile a segmenti o punti)

21

Backface Culling: quanto ci fa risparmarie?



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

22

Considerazioni generali sul Pipeline (ripetizione)

- ✓ Le tre fasi, che sono in cascata, avvengono in parallelo
  - ⇒ Mentre processo (trasformo) i vertici di un triangolo, sto anche processando (rasterizzando) il triangolo precedente, e processando (computando il lighting) dei frammenti del triangolo generato ancora prima
- ✓ Ogni fase del pipeline avviene in parallelo
  - ⇒ Ogni vertice, triangolo, frammento può essere processato in contemporanea con altri
- ✓ Il pipeline procede tanto velocemente quanto la sua fase più lenta
  - ⇒ Che è detta «**il collo di bottiglia**» (**bottleneck**)
- ✓ Terminologia per il pipeline di rendering
  - ⇒ Se il bottleneck è la fase per vertice, l'applicazione si dice «**transform limited**» o «**geometry limited**»: non riesce ad effettuare abbastanza trasformazioni geometriche (ad esempio, la mesh ha una risoluzione troppo elevata)
  - ⇒ Se è la fase per frammento, l'applicazione si dice «**fill limited**»: non riesce a riempire il buffer di pixel abbastanza velocemente (ad esempio, l'immagine di output ha una risoluzione troppo elevata)

23

Quando usare il backface culling (sommario)

- ✓ *Se valgono le ipotesi dette,* allora il back-face culling non cambia le immagini generate
  - ⇒ In caso contrario, genera artefatti (scompare il «retro» delle superfici)
- ✓ In questo caso, è conveniente perché…
  - ⇒ migliora le prestazioni delle applicazioni fill-limited, riducendo di circa il 50% il numero di frammenti prodotti e da processare
    - 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 occluse
    - Non è però sufficiente, nel caso di mesh concave, perché non rimuove *tutte* le facce auto-occluse (vedi dopo)
- ✓ E' un test eseguito in automatico dall'HW della GPU
  - ⇒ Quindi, programmando l'API a basso livello (come WebGL o OpenGL), il programmatore si limita ad abilitarlo oppure disabilitarlo

25

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;
```

27

Rimozioni delle superfici nascoste: attraverso il Depth-buffer

✓ Idea: eseguire un 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 corrente di ogni pixel…

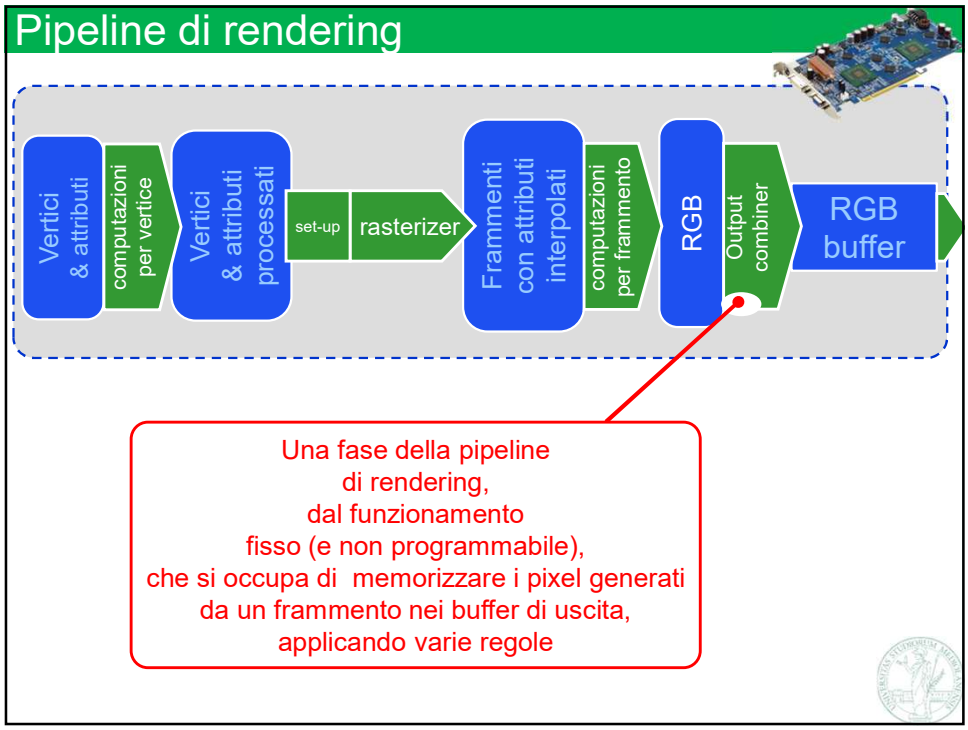
✓ «Depth-buffer»: (a volte: «Z-buffer»)

⇒un buffer 2D (una griglia, una matrice di valori)

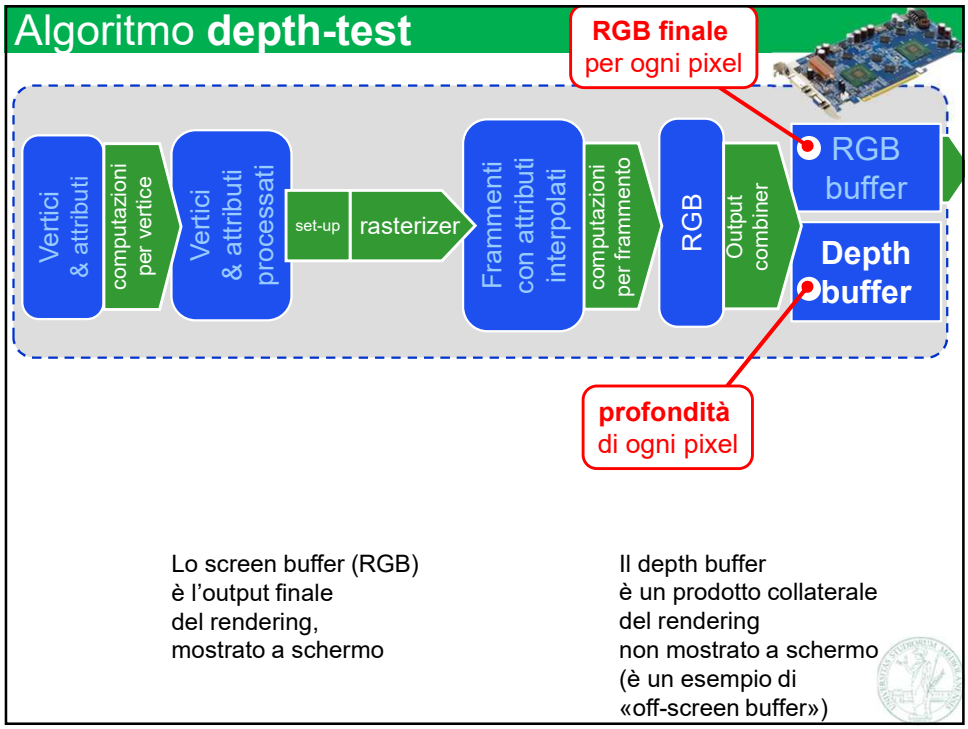
⇒stessa risoluzione dello screen buffer

⇒per ogni posizione [x , y] memorizza il depth del frammento più vicino scritto in [x , y]

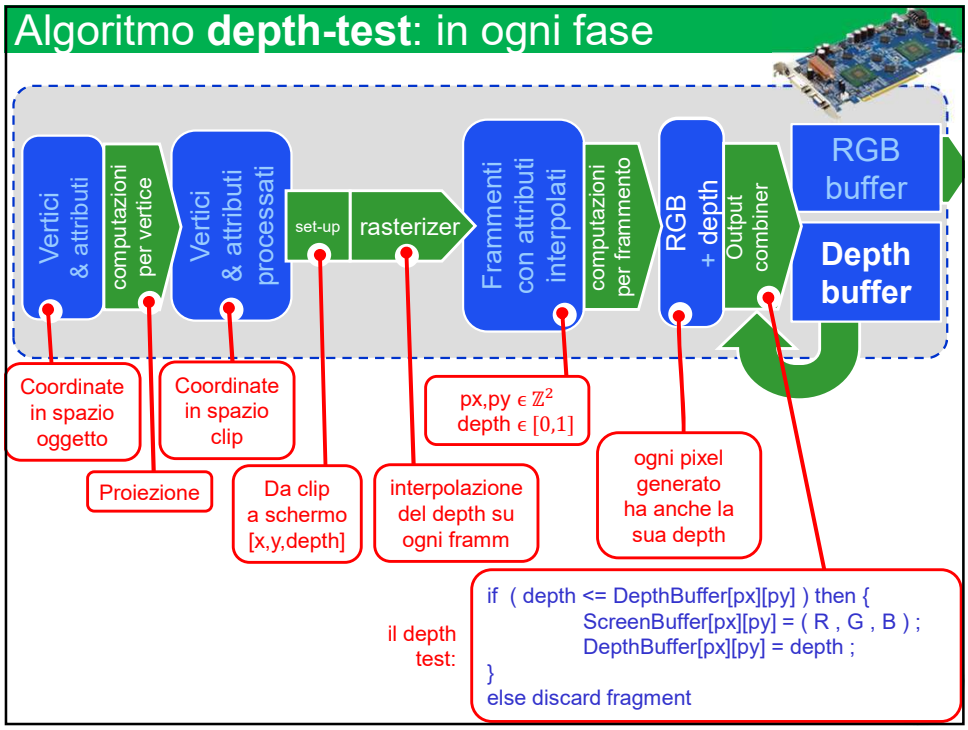
28


Pipeline di rendering
Vertici & attributi
computazioni per vertice
Vertici & attributi processati
set-up
rasterizer
Frammenti con attributi interpolati
computazioni per frammento
RGB
Output combiner
RGB buffer
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

29


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 (è un esempio di «off-screen buffer»)

30



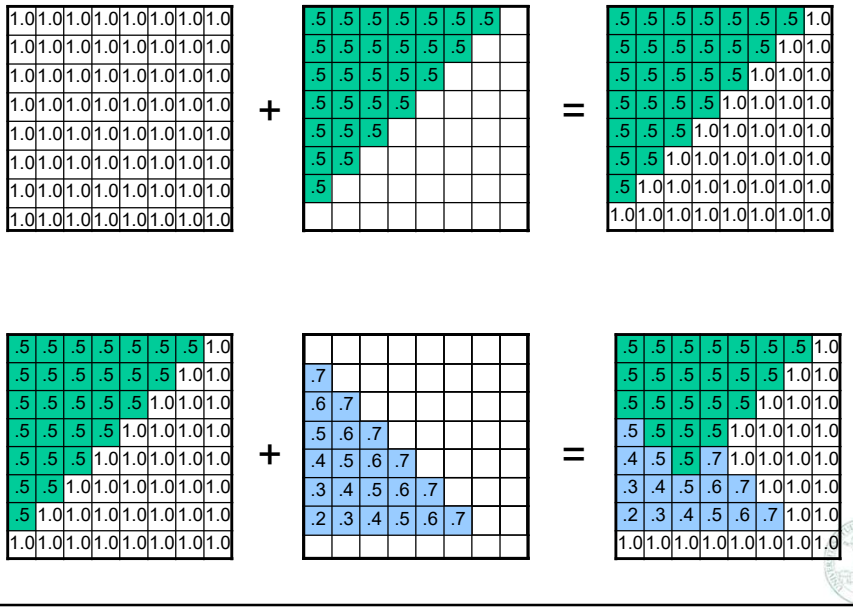
31

Valore di depth (profondità) di un frammento

- ✓ Un valore da 0 a 1
  - ⇒ 0: pixel più vicino possibile all'osserevatore
  - ⇒ 1: pixel più lontano possibile dall'osserevatore
  - ⇒ i frammenti con depth non inclusa fra 0 e 1 sono scartati automaticamente dal rasterizzatore (il pixel non è nel view frustum, è scarato dal far o near clippling plane)
- ✓ E' la coordianta Z dello spazio schermo
  - ⇒ Lo spazio in ogni pixel ha coordinate intere in x e y
  - ⇒ Vedi lezione sulla proiezione e resterizzazione
- ✓ E' interpolato, dentro ogni primitiva, per ottenere il valore di *depth* di ogni frammento
  - ⇒ Attraverso coordinate baricentriche
  - ⇒ Come qualsiasi altro valore definito sui vertici

33

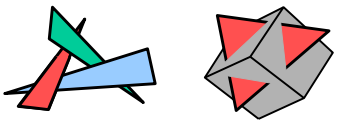
Algoritmo dello **depth-test**: esempio

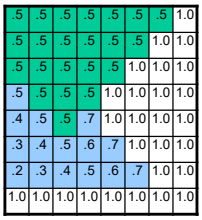


35

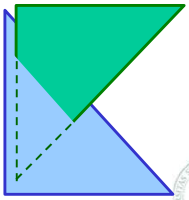
Algoritmo del **depth-test**: vantaggi

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





- ✓ Adatto all'implementazione parallela quindi HW ☺



36

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

- ✓ Costa un po' di memoria (GPU)
  - ⇒ per memorizzare lo il depth buffer
  - ⇒ 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 tardi nel pipeline
  - ⇒ quando tutta la computazione precedente 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 HW mettono in atto molte ottimizzazioni)
  - ⇒ per questo, il depth test è gestito da una apposita fase non programmabile
    (a valle delle computazioni programmabili per frammento) detta output combiner

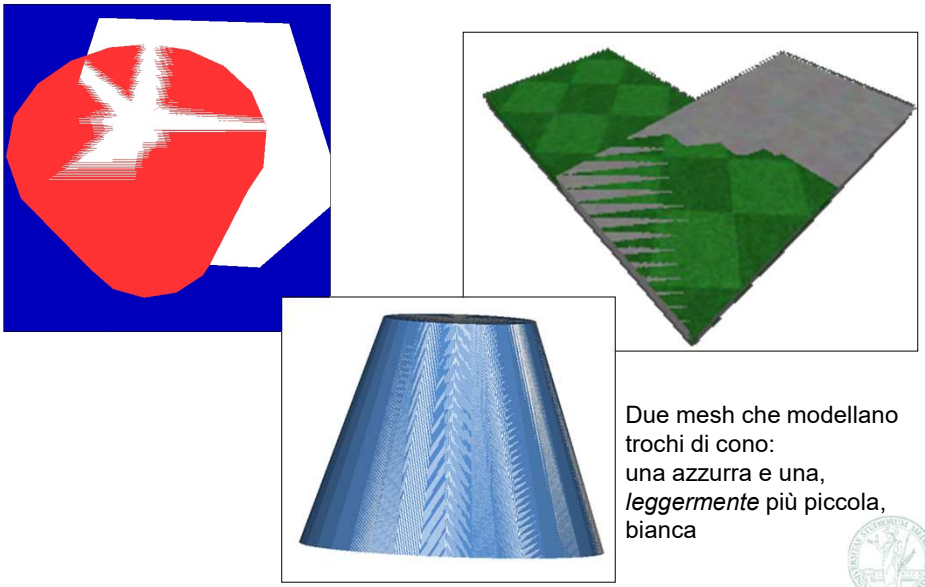
39

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 $d \in \mathbb{Z}$ (da 0 a 255)
  per rappresentare il razionale $d \,/\, 255 \in \mathbb{Q}$
- ✓ La precisione necessaria al test impone di usare più di 8 bit:
  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 coincidenti)

40

Z-figthing: esempi

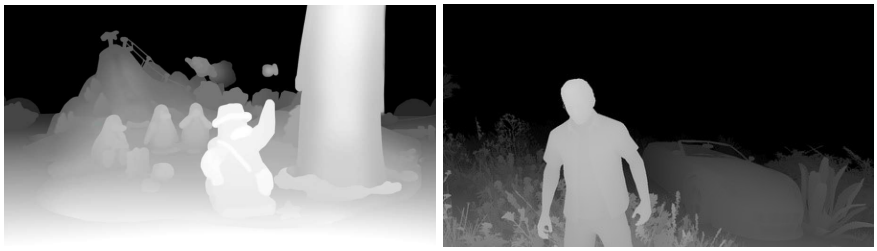


Due mesh che modellano trochi di cono:
una azzurra e una, *leggermente* più piccola, bianca

41

Effetto collaterale del Depth test: ogni rendering produce anche una depth image

- ✓ Dopo ogni rendering,
  ho prodotto non solo un'immagine a colori (RGB per pixel)
  ma anche un depth-buffer (profondità per pixel)
  - ⇒ una depth image che rappresenta un «bassorilievo» della scena
  - ⇒ Si tartta quindi di una range scan «virtuale»



- ⇒ 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

44

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: è lo stesso!
  - ⇒ L'efficienza può essere però diversa:
    scartare è più efficiente di
    scrivere-per-poi-sovrascrivere
    - Quindi il depth sorting è ancora utile, ma solo come ottimizzazione
    - L'ordine ottimale è front-to-back
    - L'opposto di quello usato nell'algoritmo del pittore!

45

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
  - ⇒ 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

49