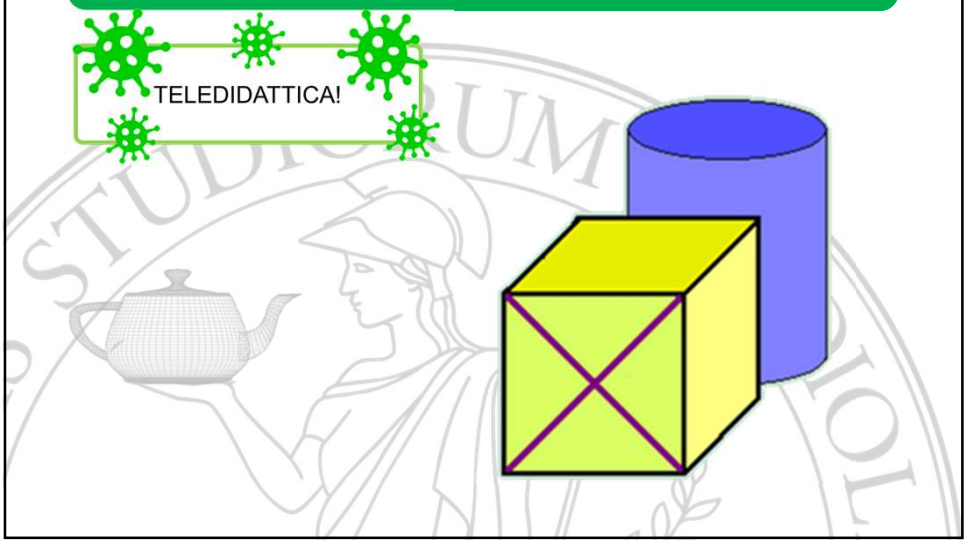
Marco Tarini - Computer Graphics 2020/2021
Università degli Studi di Milano

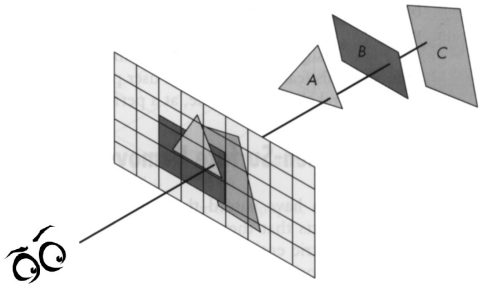
Rimozione Superfici Nascoste



1

Rimozione delle superfici nascoste (occluse)

- ✓ Gli oggetti più vicini devono coprire quelli più lontani
  - ⇒ gli oggetti lontani sono "occlusi" da quelli più vicini



Con raytracing era ovvio!
Con raserization?

2

Rimozioni delle superfici nascoste

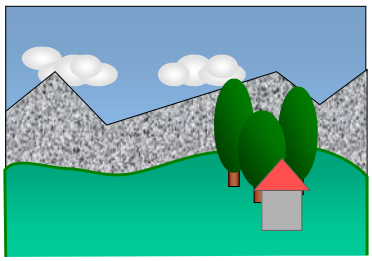
✓ Soluzione 1: basata su ordinamento

⇒ 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"



3

Algoritmo del pittore (o depth sorting)

✓Data una scena (composta da primitive)

⇒ordinare le primitive back-to-front

- cioè in ordine di crescente di z… in spazio vista!

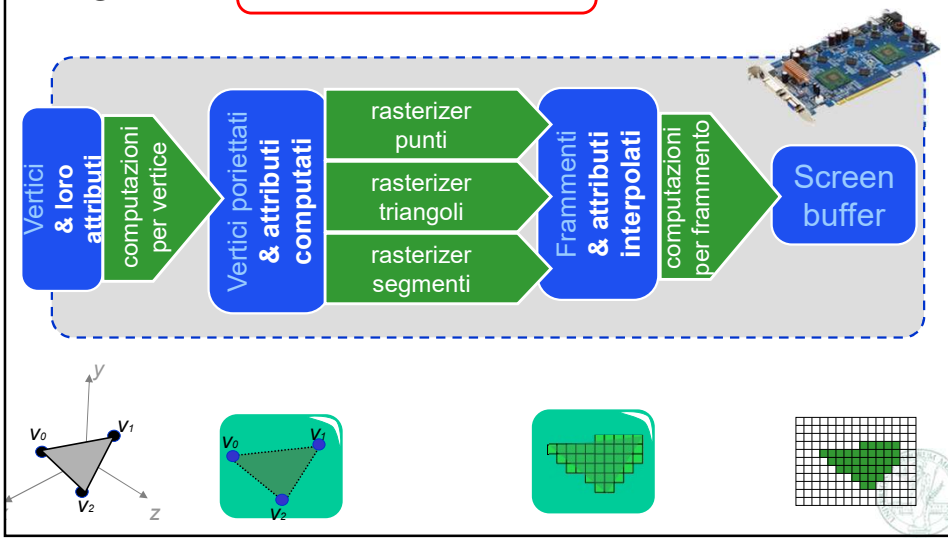
⇒rasterizzarle in successione

4

Algoritmo del pittore (depth sorting)

DOVE?

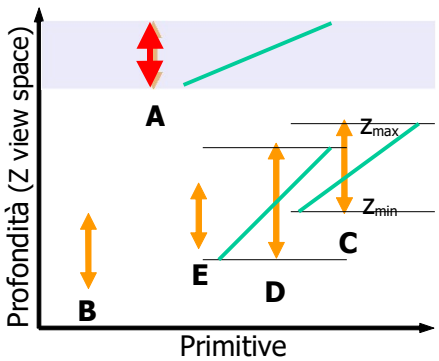
prima di mandare i triangoli.
Fatto dalla CPU ! ☹☹☹



5

Algoritmo del pittore (depth sorting)

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

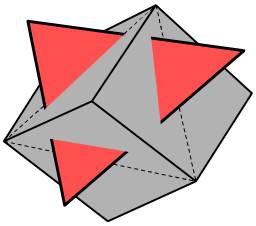


6

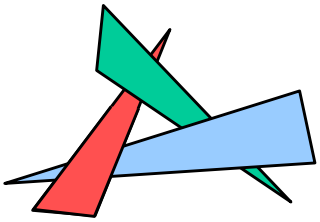
Algoritmo del pittore (depth sorting)

✓ Esiste sempre un ordine corretto?

⇒NO.
Controesempio: intersezioni



⇒NO.
Controesempio: cicli



7

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, peggio di lineare è mal tollerato

⇒Problema 2: quando effettuare l’ordinamento?
- le coordinate in spazio vista (comreso la Z)
  sono note solo dopo la trasfromazione (fino allo spazio vista)

⇒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

8

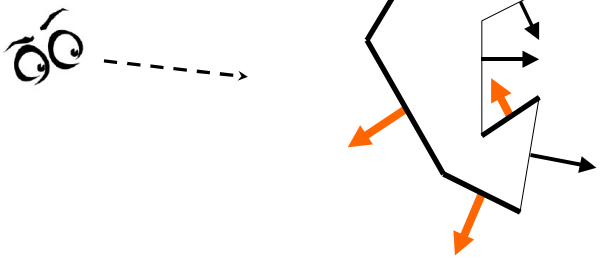
Algoritmo del pittore (depth sorting)

- ✓ 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

9

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!



10

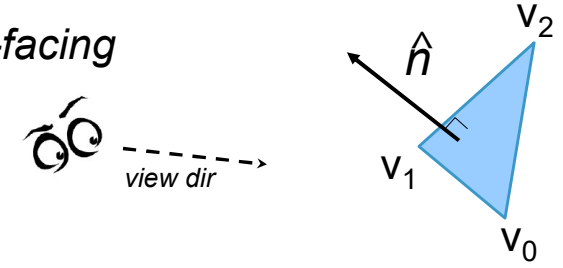
Caso particolare: mesh chiuse e ben orientate

- ✓ Back-face Culling
  - ⇒ "culling" (di una primitiva) = scarto della primitiva in fase di rendering
- ✓ Idea:
  - ⇒ ipotesi: superficie **chiusa**, **opaca** e **ben orientata**
  - → non vedrò mai l'interno!
  - → qualunque faccia si veda "da dietro" (back-facing triangles) finirà per forza *occlusa* alla vista da (almeno) un layer di facce "viste da davanti" (front-facing triangles)
  - → posso scartare tutti i triangles che sono back-facing

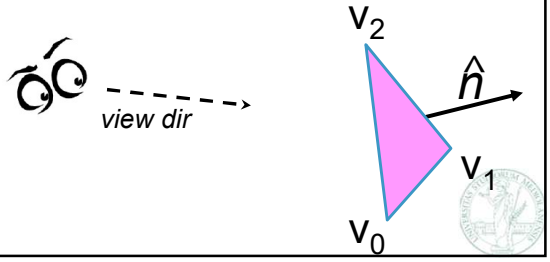
11

Caso particolare: mesh chiuse e ben orientate

- ✓ Traingolo *front-facing*



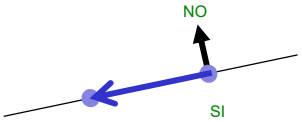
- ✓ Traingolo *back-facing*
  - ⇒ "visto di spalle"

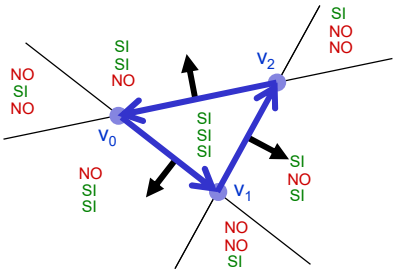


14

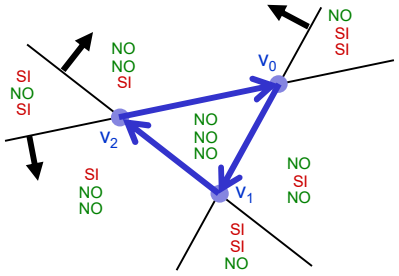
Test di appartenenza ad un triangolo

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





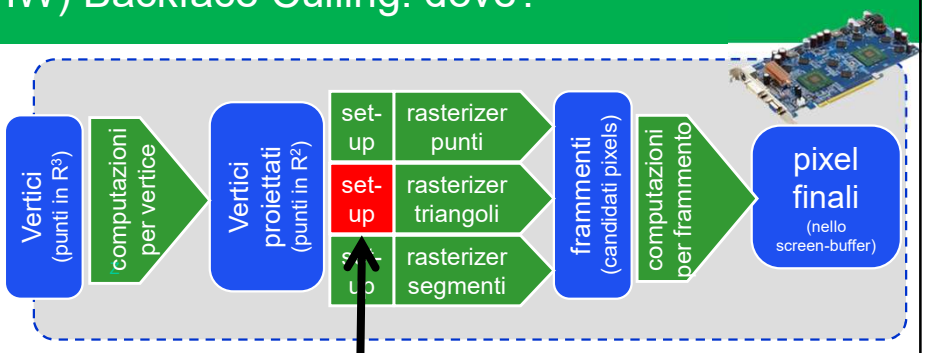
Triangolo 2D front facing
(in spazio Clip)



Triangolo 2D back facing
(in spazio Clip)

15

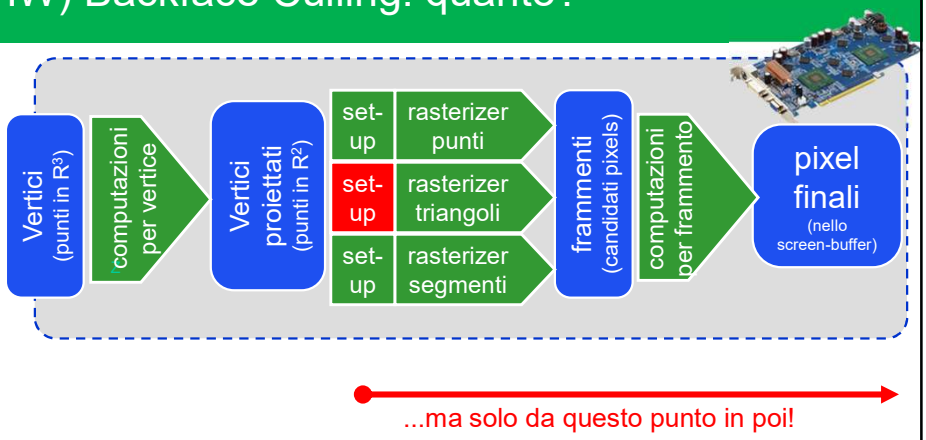
(HW) Backface Culling: dove?



QUI.
(nella fase di set-up che precede la rasterizzazione di ciascun triangoli)

16

(HW) Backface Culling: quanto?



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

17

Quando usare il backface culling

- ✓Se la mesh è chiusa è ben orientata:
  - ⇒Back-face culling: si.
  - ⇒Se la camera è dentro la mesh, scartare le facce front-facing
  - ⇒Se la camera è furi dalla mesh, (oppure se la mesh ha tutte le normali invertite): scartare le facce back-facing
- ✓Se la mesh è chiusa è ben orientata:
  - ⇒Back-face culling non può essere usato (per non incorrere in artefatti: oggetti non visibili visti da dietro)

19

Nelle API a basso livello, come WebGL

✓ Attivare e disattivare, prima di renderizzare ogni 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

20

Nelle lib più ad alto livello, come Three.js

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

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

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

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

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

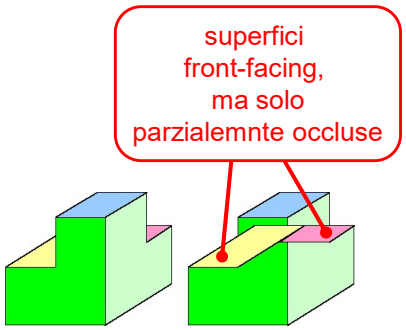
⇒Back-Face Culling disabilitato:

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

21

Rimozioni delle superfici nascoste

- ✓Back-face culling non basta a rimuovere tutte le superfici nascoste
  - ⇒esempio:



22

Rimozioni delle superfici nascoste con 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

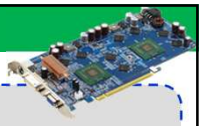
23

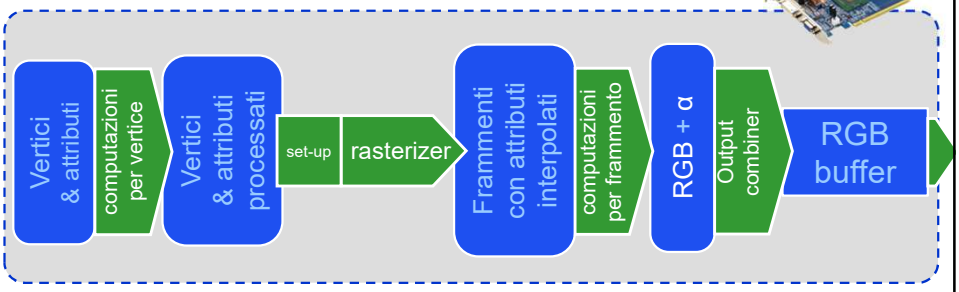
Valore di depth (profondità) di un frammento

- ✓ Un valore da 0 a 1
  - ⇒ 0: pixel più vicino possible
  - ⇒ 1: pixel più lontano possible
- ✓ Calcolato (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 “varying” definite sui vertici
  - ⇒ Coordinate baricentriche

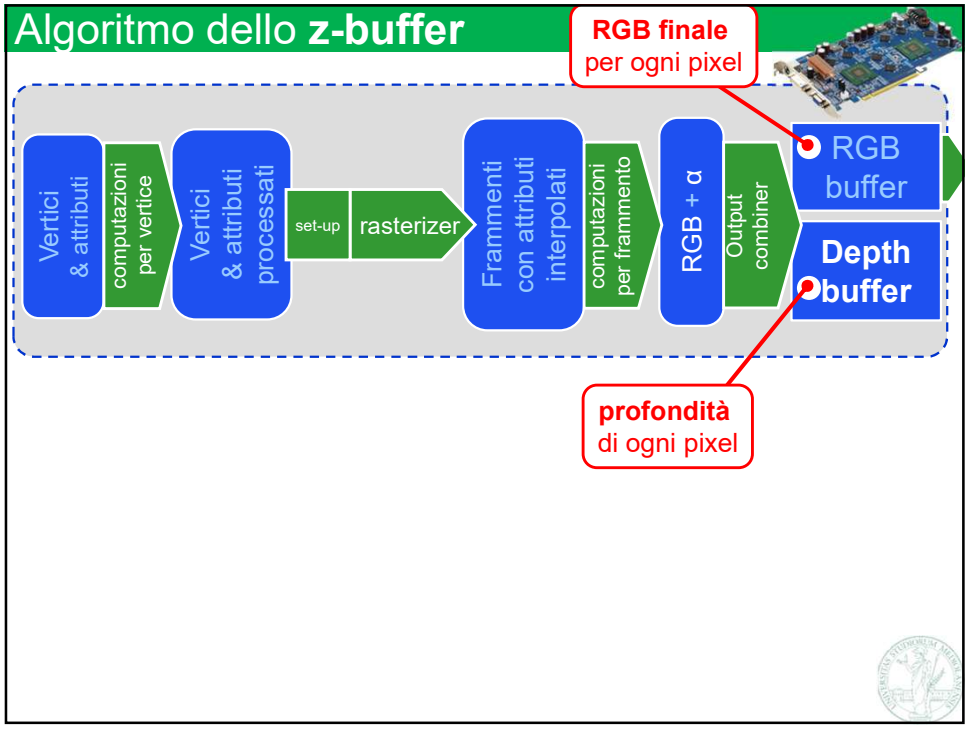
24

Algoritmo dello **z-buffer**

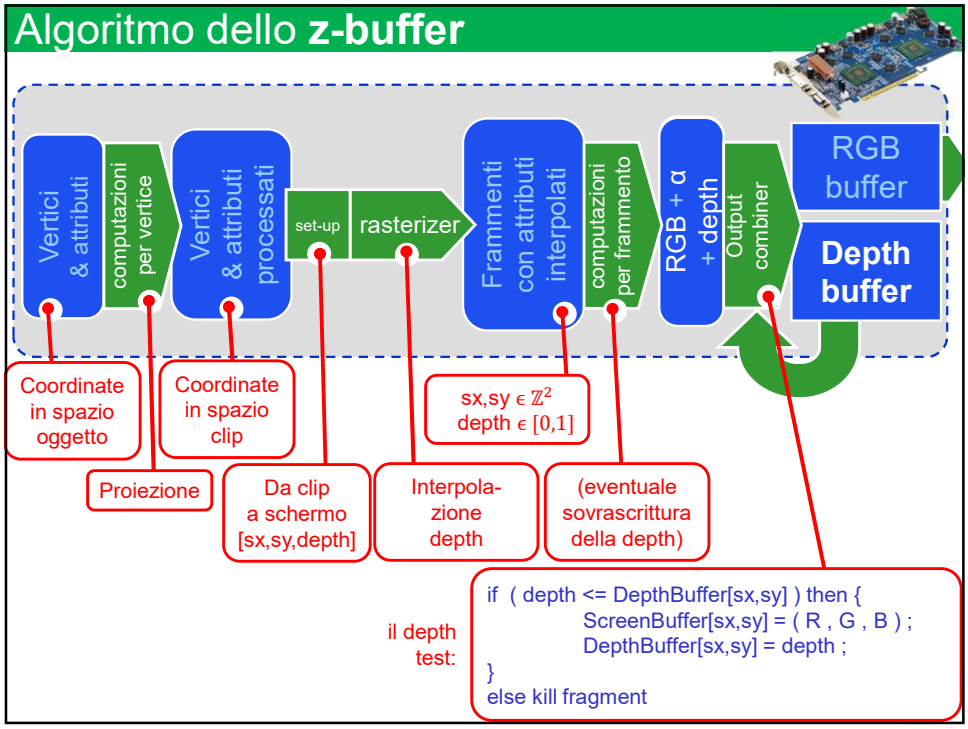




25


Algoritmo dello z-buffer
RGB finale
per ogni pixel
Vertici & attributi
computazioni per vertice
Vertici & attributi processati
set-up
rasterizer
Frammenti con attributi interpolati
computazioni per frammento
RGB + α
Output combiner
RGB buffer
Depth buffer
profondità
di ogni pixel

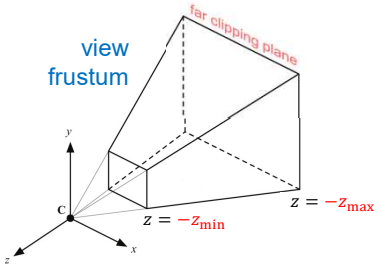
26

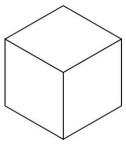

Algoritmo dello z-buffer
Vertici & attributi
computazioni per vertice
Vertici & attributi processati
set-up
rasterizer
Frammenti con attributi interpolati
computazioni per frammento
RGB + α + depth
Output combiner
RGB buffer
Depth buffer
Coordinate in spazio oggetto
Coordinate in spazio clip
sx,sy ϵ ℤ²
depth ϵ [0,1]
Proiezione
Da clip a schermo [sx,sy,depth]
Interpola-zione depth
(eventuale sovrascrittura della depth)
il depth test:
if (depth <= DepthBuffer[sx,sy]) then {
ScreenBuffer[sx,sy] = (R , G , B) ;
DepthBuffer[sx,sy] = depth ;
}
else kill fragment

27

Limiti dello spazio visibile (inquadrato)

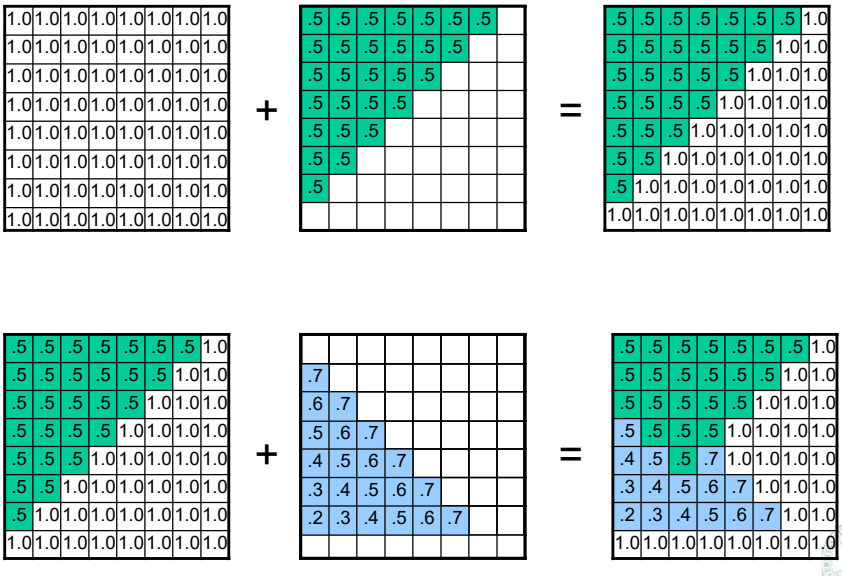
		Spazio Vista	Spazio Clip	Spazio Schermo	
x	da:	«left clipping plane»	-1	0	pixel coordinates
	a:	«right clipping plane»	$+1$	res_x	
y	da:	«bottom clipping plane»	-1	0	
	a:	«top clipping plane»	$+1$	res_y	
z	da:	$-z_{min}$	-1	0	pixel depth
	a:	$-z_{max}$	$+1$	1	





28

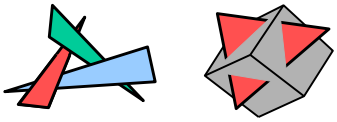
Algoritmo dello **z-buffer:** esempio



30

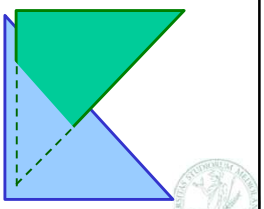
Algoritmo dello **z-buffer:** vantaggi

- ✓ 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 ☺



31

Algoritmo dello **z-buffer:** 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*
  - ⇒ causa detto: “z-fighting”
  - ⇒ quando la precisione non è sufficiente
    es., se si renderizzano due superfici parallele molto vicine
- ✓ I frammenti vengono scartati solo alla fine del pipeline
  - ⇒ tutta la computazione è già stata inutilmente effettuata
- ✓ Assume superfici del tutto opache
  - ⇒ problemi con le superfici semitrasparenti
- ✓ Dati condivisi in lettura e scrittura ☹
  - ⇒ complicazione per chi implementa HW
  - ⇒ efficienza ameno in parte impattata (ma test molto ottimizzato)
  - ⇒ è un test **HardWired in GPU** (vediamo come)

32

Algoritmo dello z-buffer

✓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

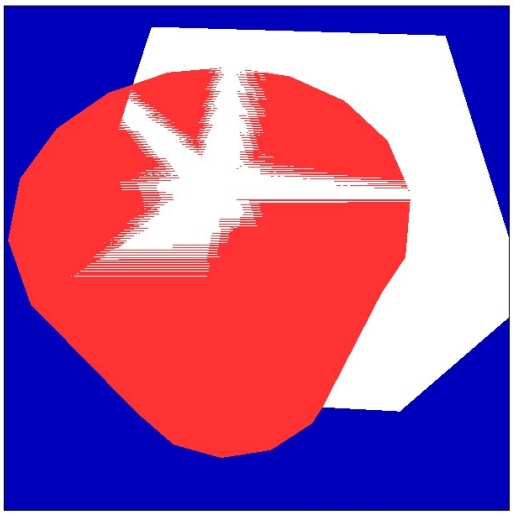
⇒risultato finale: lo stesso!
(eccetto i rarissimi casi di pareggio)

⇒L'efficienza può variare:
disegnare poi sovrascrivere è meno efficiente
di scartare

- Quindi, qrdine ottimale: front-to-back

33

Z-figthing



34

Il rendering produce anche depth images

- ✓ Effetto collaterale dell'algoritmo di depth buffer: 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

  - ⇒ Il buffer RGB viene mandato a schermo,
  - ⇒ il depth-buffer di solito viene semplicemente scartato
  - ⇒ …ma alcuni algorimi lo sfruttano invece per gli scopi più vari

36

Depth test: alcune considerazioni sull'efficienza

- ✓ Il depth test è hard-wired affinché sua implementazione sia ottimizzata in GPU
  - ⇒ quindi nell'output combiner, non nel fragment shader programmabile!
  - ⇒ necessario, perché il depth-test richiede lettura + scrittura in memoria condivisa (il Depth Buffer): problema per il parallelismo!
- ✓ Molte ulteriori ottimizzazioni sono fatte in HW (silenziosamente) per consentire
  - ⇒ di anticipare il test (early depth test) e/o
    - regola generale per i test che scartano: prima vengono fatti meglio è
  - ⇒ di testare gruppi di frammenti alla volta, o intere primitive (hierarichical Z buffer)

37