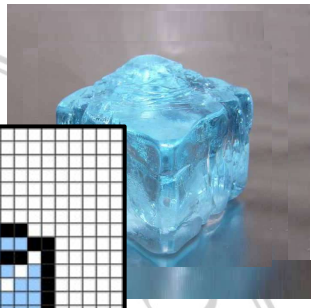
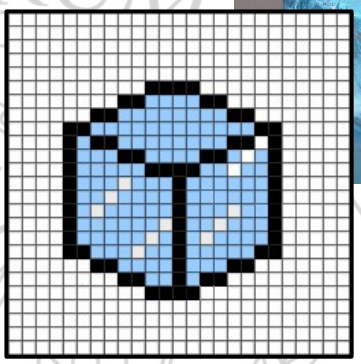
Marco Tarini - Computer Graphics 2023/2024
Università degli Studi di Milano

Rasterizing primitives

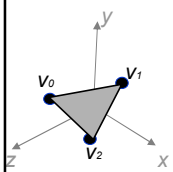


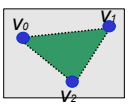


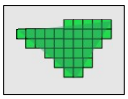
1

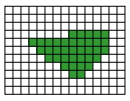
Rasterization based pipeline

Vertici (punti in R^3) → computazioni per vertice → Vertici proiettati (punti in R^2) → RASTERIZ-ZAZIONE → frammenti (candidati pixels) → computazioni per frammento → screen buffer









2

Rasterizzazione

- ✓ Individuare i frammenti che compongono una primitiva
  - ⇒ Uno per ogni pixel coperto dalla primitiva
  - ⇒ I frammenti prodotti vengono passata alla fase successiva (che determina il colore RGB del pixel)
- ✓ E' un procedimento puramente 2D
- ✓ E' estremamente parallelizzabile
  - ⇒ Per sfruttare questo, si adotta un hardware parellelo
  - ⇒ Sulle GPU, è implementato in hardware (architettura specializzata per lo scopo)
  - ⇒ E' una fase dell'hardware molto efficiente e poco flessibile (non è «programmabile» dall'utente) è «hard-wired» in pratica: fa sempre la stessa cosa,
  - ⇒ hardware diversi implementano algoritmi diversi (spesso)

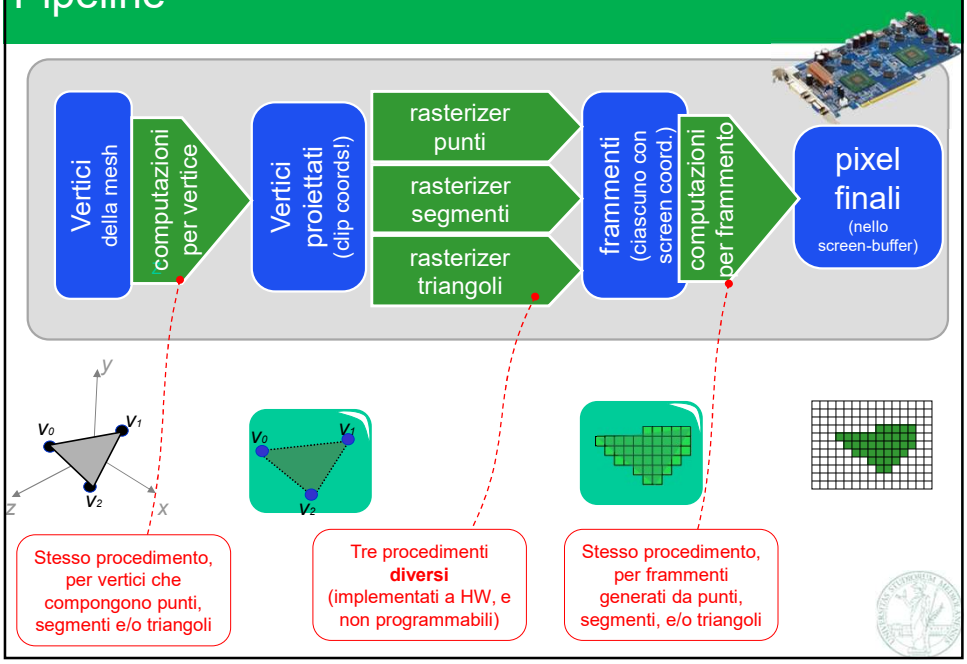
3

Rasterizzazione

- ✓ Gli hardware GPU attuali sono pesati per rasterizzare solo tre tipi di primitive:
  - ⇒ Triangoli 2D – 3 vertici
  - ⇒ Segmenti 1D («linee») – 2 vertici
  - ⇒ Punti 0D («point splats») – 1 vertice
- ✓ Ovviamente, il primo caso è particolarmente utile ed ottimizzato
- ✓ I vertici sono passati al rasterizzatore in coordinate clip omogenee
- ✓ Per prima cosa, vengono convertite in coordinate schermo cartesiane
  - ⇒ come sappiamo già
- ✓ Poi, il rasterizzatore usa solo le x e la y (in spazio schermo, cioè le coordinate pixel),
  - ⇒ la z (cioè la *depth* del pixel) viene ignorata in questa fase

4

Pipeline



7

Algoritmo di rasterizzazione di triangoli

- ✓ Vanno individuati i frammenti che compongono i triangoli
  - ⇒ *Input:* tre punti 2D (le posizioni x,y dei vertici in screen space)
  - ⇒ *Output:* i frammenti interni al triangolo
- ✓ Un frammento per ogni pixel dello screen buffer interno al triangolo
  - ⇒ Ogni frammento prodotto ha coordinate schermo intere
- ✓ Devono essere prodotti tutti e soli i frammenti le cui coordinate cascano all'interno del triangolo 2D
- ✓ Nota: i tre vertici di input hanno coordiante (in generale) non intere
  - ⇒ Sono le coordiante x e y risultanti dalle trasformazioni spaziali

8

Algoritmo di rasterizzazione di triangoli

✓ Il rasterizzatore si occupa anche di
INTERPOLARE GLI ATTRIBUTI
in ogni frammento prodotto

⇒ *Input ulteriore:* un set di attributi per ogni vertice
(qualsiasi insieme di vettori, scalari…)

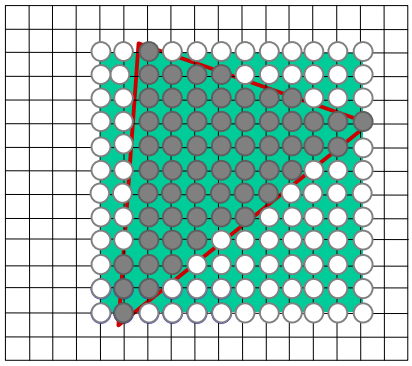
⇒ *Output ulteriore:* un set di valori interpolati,
per ogni frammento prodotto

⇒ come sappiamo, l'output è dato da
l'iterpolazione degli input
avente come pesi
le coordinate baricentriche
del frammento nel triangolo renderizzato

⇒ Gli attributi interpolati sono passati dal rasterizzatore
al processing per frammento

9

Esempio di algoritmo per rastizzazione di triangoli (parallelizzabile!)



1. Trovo un rettangolo
(di coordianate intere)
che contiene il triangolo
detto "bounding box"
o "bounding rectangle"

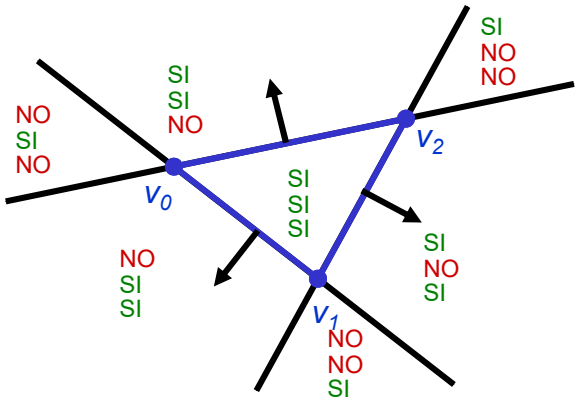
2. Processo tutti
le posizioni interne
(nota: questo è parallelizzabile)

3. Per ogni posizione interna:
se è dentro al triangolo,
allora produco un frammento

10

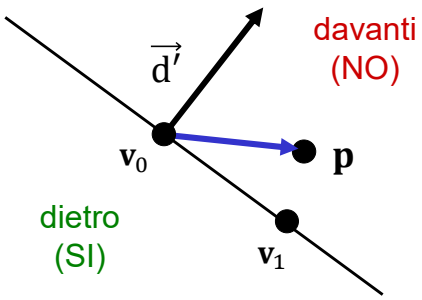
Test di appartenenza ad un triangolo

- ✓ Triangolo 2D = interesezione di 3 semipiani
- ✓ Un punto 2D è interno al triangolo 2D *sse* appartiene a tutti e tre i semipiani



11

Test di appartenenza ad un semipiano (ci serve solo in 2D)



Da quale parte della retta passante per $\mathbf{v}_0$ e $\mathbf{v}_1$ si trova il punto $\mathbf{p}$?

Soluzione:

$$\vec{d} = (\mathbf{v}_1 - \mathbf{v}_0) = \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

$$\vec{d}' = \begin{pmatrix} -d_y \\ d_x \end{pmatrix}$$

il vettore $\vec{d}'$ è $\vec{d}$ ruotato di 90°

la cosiddetta *Edge function* dell'edge da $\mathbf{v}_0$ a $\mathbf{v}_1$

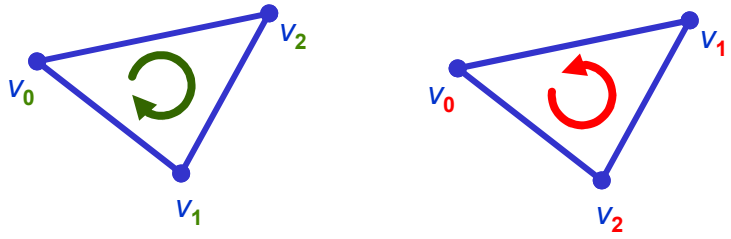
Basta verificare se:

$$(\mathbf{p} - \mathbf{v}_0) \bullet \vec{d}' < 0$$

12

Domande avanzate

1. Cosa succede a questo algoritmo se l'orientamento del triangolo è in verso opposto? Come puoi riconoscere i frammenti interni al tri in entrambi i casi?



2. Date le coordinate 2D di $v_1\ v_2\ v_3$ sapresti calcolare le coordinate baricentriche del frammento in posizione **p**? (sono necessarie per interpolare gli attributi)

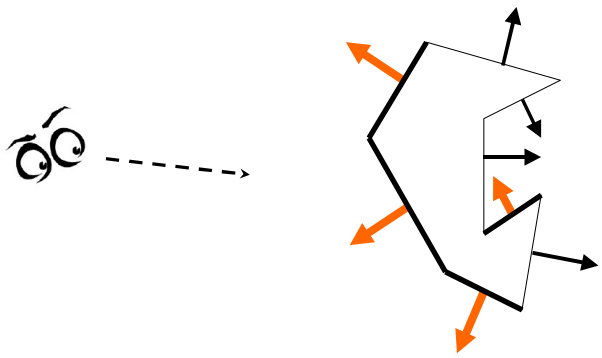
13

Nel caso di mesh chiuse e ben orientate: Backface Culling

✓ Concetto:

⇒ se una superficie **chiusa** e **opaca**…

→ non vedrò mai l'interno!



14

Backface Culling

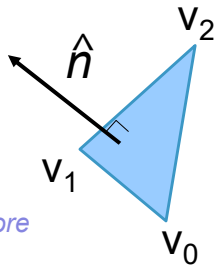
- ✓ Back-face Culling
  - ⇒ "to cull" = scartare.
    To cull a face = scartare la primitiva in fase di rendering
- ✓ Idea:
  - ⇒ ipotesi: mesh **ben orientata** e **chiusa**,
  - ⇒ ipotesi: POV esterno alla superficie
  - ⇒ ipotesi: materiale opaco (nel senso di non trasparente)
  - → non vedrò mai 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 triangoli che sono back-facing,
    (sapendo che saranno automaticamente occlusi)

15

Back-face culling: un'utile ottimizzazione

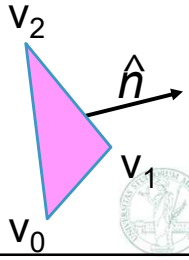
- ✓ Traingolo *front-facing*
  - ⇒ "visto da davanti"

direzione verso l'osservatore



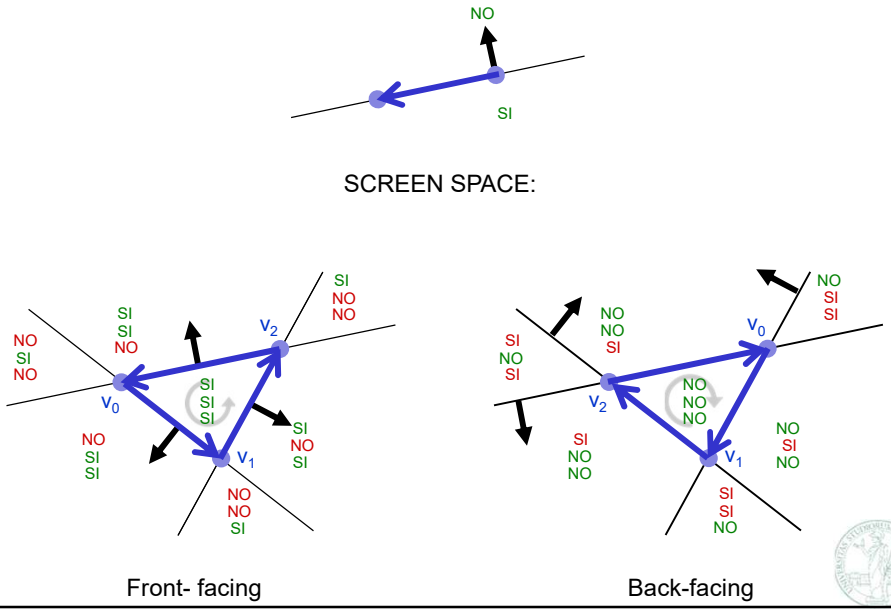
- ✓ Traingolo *back-facing*
  - ⇒ "visto di spalle"

direzione verso l'osservatore



16

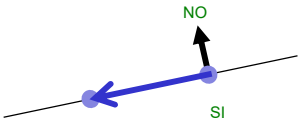
Back-face culling e test di appartenenza di frammento a triangolo

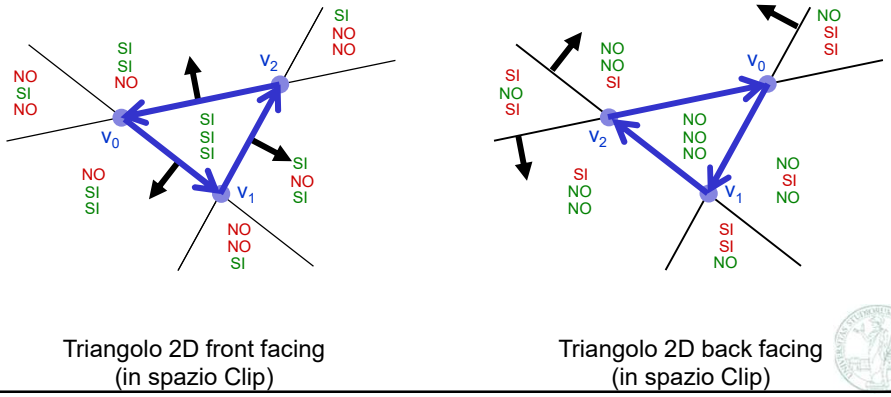


17

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

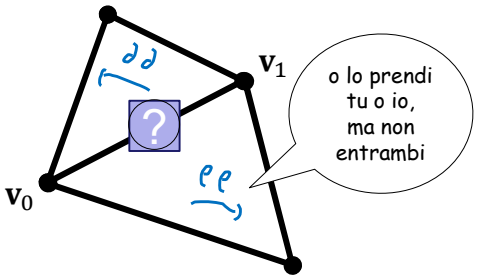




18

Dettaglio: cosa succede se il frammento è "proprio sulla linea"? (cioè se la edge function fa proprio 0)

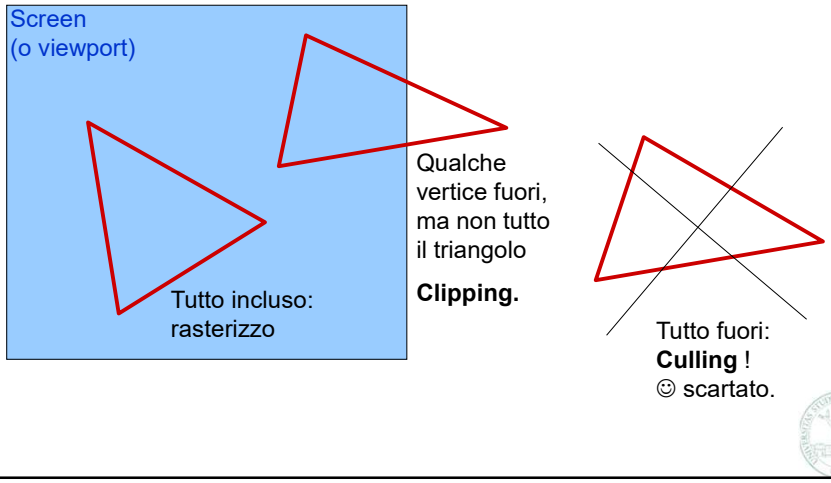
- ✓ Molte soluzioni sono possibili, ma una regola deve sempre valere (è compito dell'implementazione hardware farla valere):
- ✓ un frammento su una linea "a cavallo" fra due triangoli deve essere rasterizzato da *esattamente* uno dei due
  - ⇒ Non entrambi: perché vanno evitati gli "overdraw" inutili
  - ⇒ Non nessuno dei due: perché vanno evitati i gap (buchi) fra i i due



- ✓ Cioè: se il frammento è scartato dell'edge "da $\mathbf{v}_0$ a $\mathbf{v}_1$" allora NON deve essere scartato testando l'edge in senso opposto "da $\mathbf{v}_1$ a $\mathbf{v}_0$", e viceversa

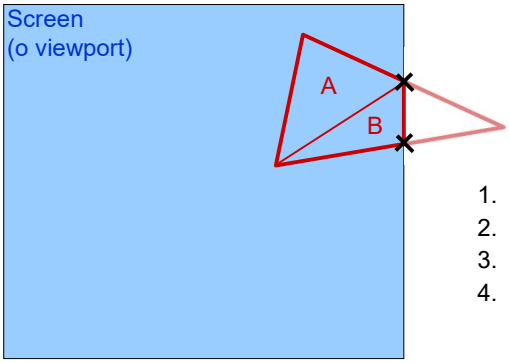
20

Clipping e culling



21

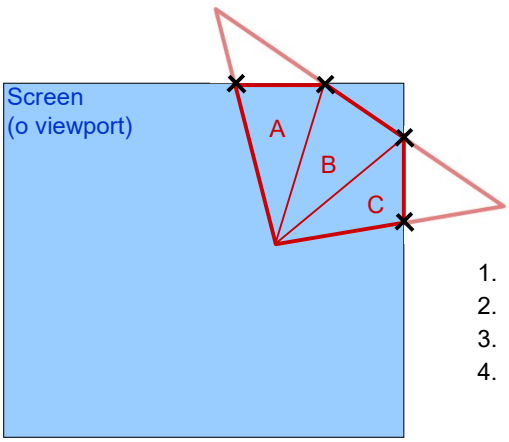
Clipping: la vecchia scuola



1. Trovo intersezioni
2. Unisco intersezioni
3. Divido poligono in triangoli
4. Rasterizzo ogni triangolo
   A e B

22

Clipping: la vecchia scuola



1. Trovo intersezioni
2. Unisco intersezioni
3. Divido poligono in triangoli
4. Rasterizzo ogni triangolo
   A, B, e C

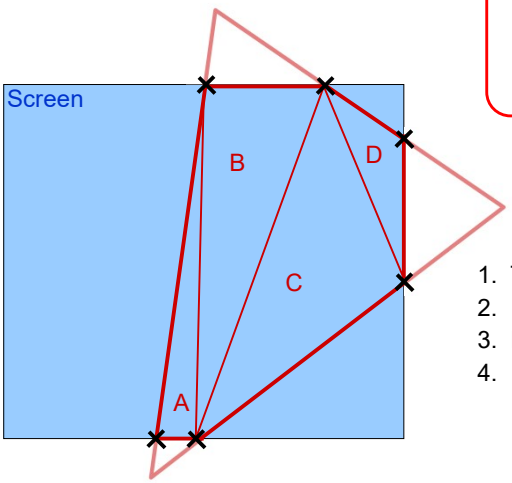
23

Clipping: la vecchia scuola

Caso pessimo molto complicato

Non adatto ad
una implementazione HW

L'HW deve prevedere il
caso pessimo, anche se è raro

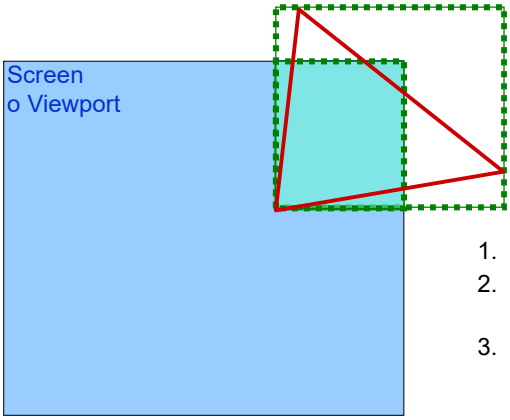


1. Trovo intersezioni
2. Unisco intersezioni
3. Divido poligono in triangoli
4. Rasterizzo ogni triangolo
   A,B,C,D

24

Clipping: versione più adatta ad una implementazione HW

✓Clipping

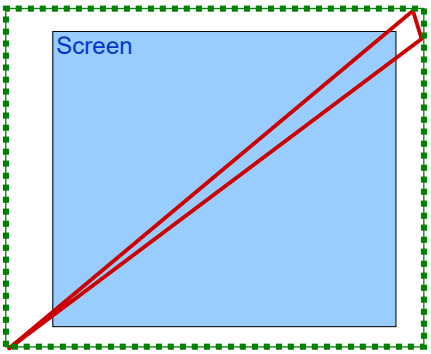


Come si interseca
un bounding box
con lo schermo
(cioe' col rettangolo
definito dal ViewPort)?

1. Trovo bounding box
2. Interseco bounding box
   con schermo (o viewport)
3. Rasterizzo nel bounding box
   come normale

25

Un caso molto sfavorevole



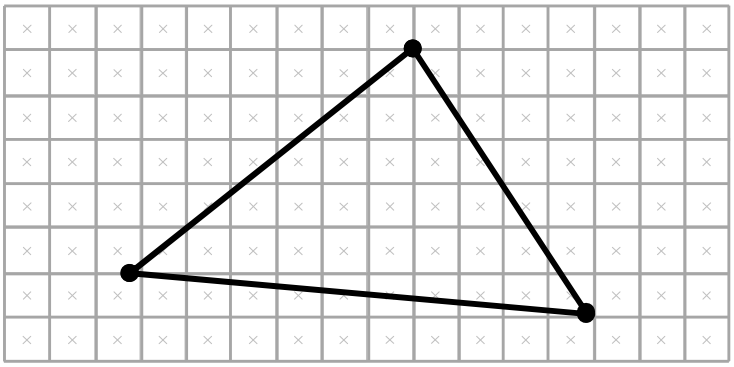
Per questo motive
(e molti altri)
i triangoli lunghi e stretti
non sono ideali

Triangoli:
- lunghi e stretti
- orientati lungo la direzione diagonale dello schermo (in spazio clip)

necessino il testing di *ogni* triangolo sullo schermo

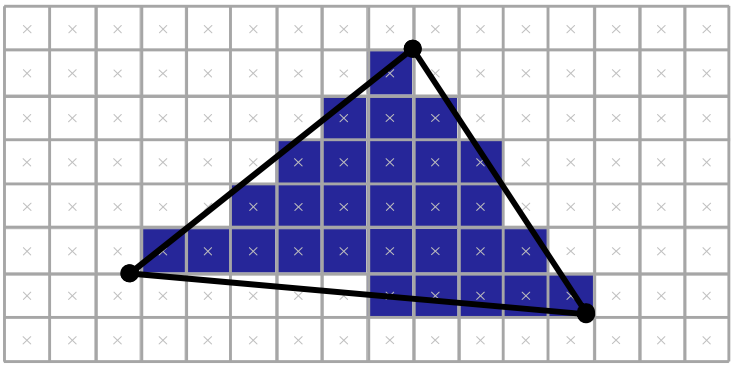
29

Rasterizzare Triangoli



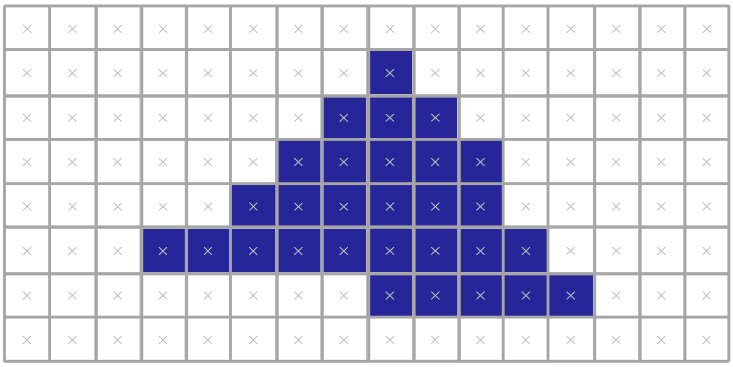
30

Rasterizzare Triangoli



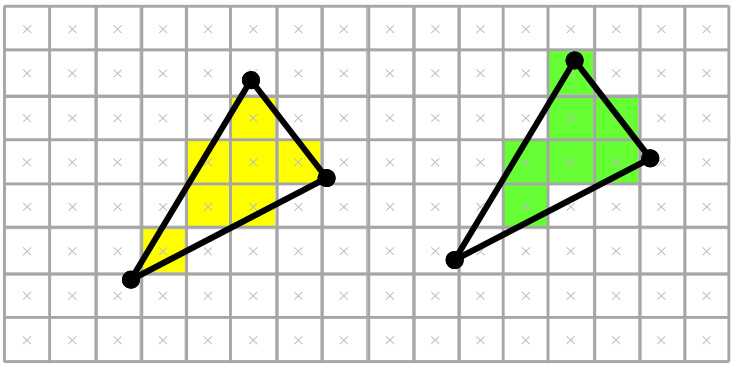
31

Rasterizzare Triangoli



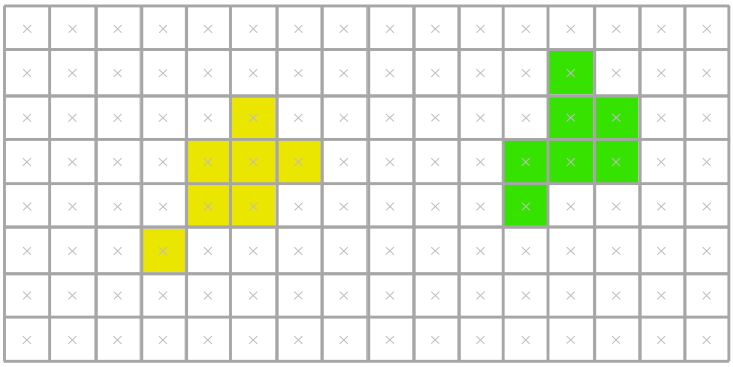
32

Rasterizzare Triangoli



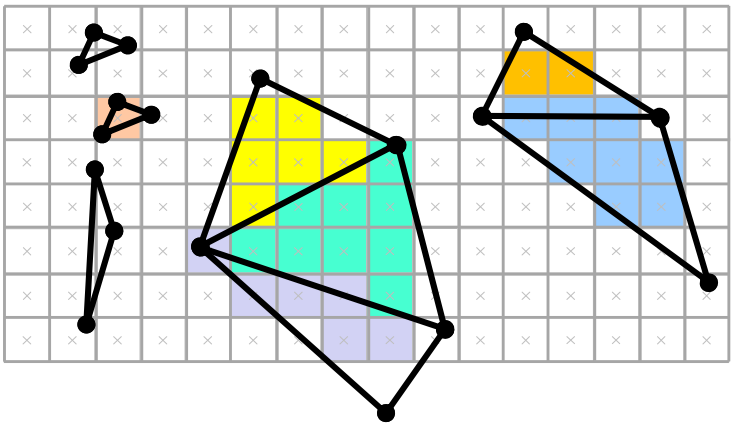
36

Rasterizzare Triangoli



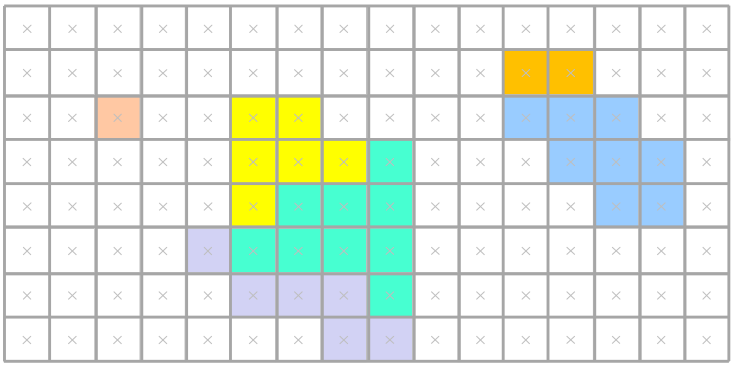
37

Rasterizzare Triangoli



47

Rasterizzare Triangoli



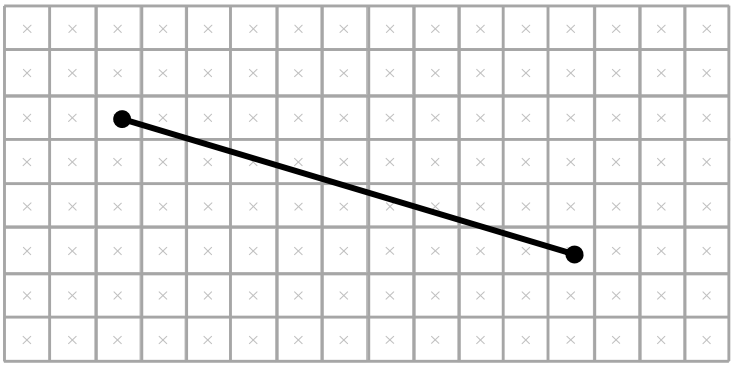
48

Note: rasterizzazione triangoli 1/2

- ✓ Triangoli della stessa forma ma posizioni diverse possono generare insieme di frammenti diversi
- ✓ Un triangolo può generare qualsiasi numero di frammenti
  - ⇒ anche nessuno!
  - ⇒ anche tutti quelli sullo schermo
- ✓ Se un triangolo è parzialmente fuori al viewport, vengono generati solo i frammenti interni al viewport («clipping»)

49

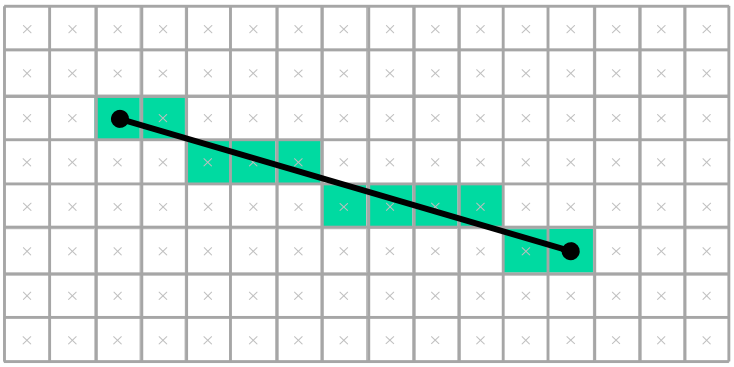
Rasterizzare Linee



Bresenham's line algorithm (1962)

51

Rasterizzare Linee



Bresenham's line algorithm (1962)

52

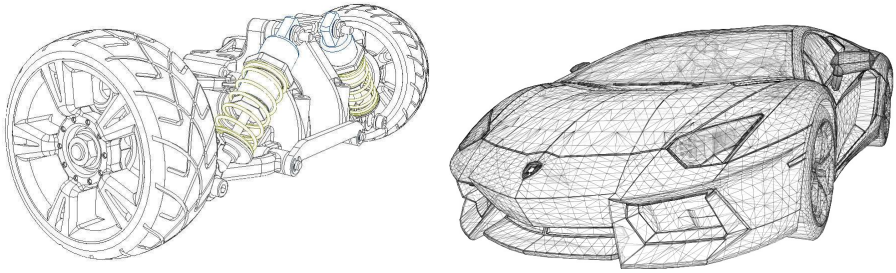
Nota storica: Bresenham's line algorithm

- ✓ Un algoritmo iterativo per rasterizzare line
- ✓ Vantaggi:
  - ⇒ Richiede solo computazioni fra interi (no virgola mobile)
  - ⇒ Efficienza
- ✓ Svantaggi:
  - ⇒ intrinsecamente sequenziale (difficilmente parallelizzabile)
  - ⇒ ogni iterazione dipende dalla precedente ☹
- ✓ Molto usato in passato, non più oggi (almeno, non nelle GPU)

53

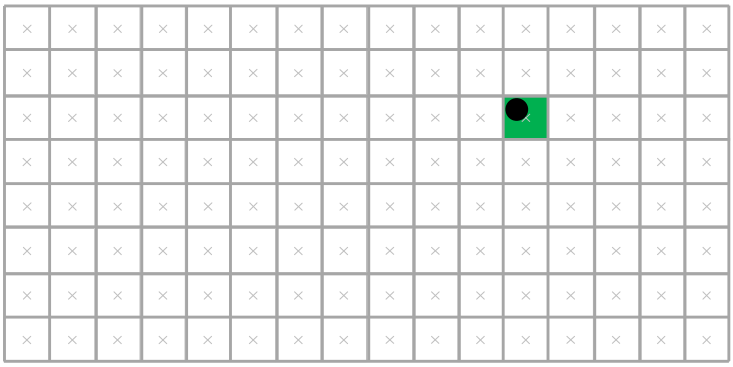
Rasterizzare Linee

✓La rasterizzazione delle linee è usata nella modalità di rendering detta *wireframe*



54

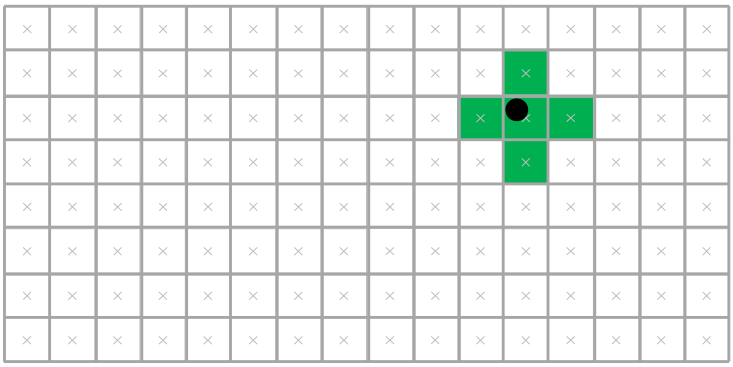
Rasterizzare punti



Point “splat”

56

Rasterizzare punti



Point “splat”
di dimensione maggiore

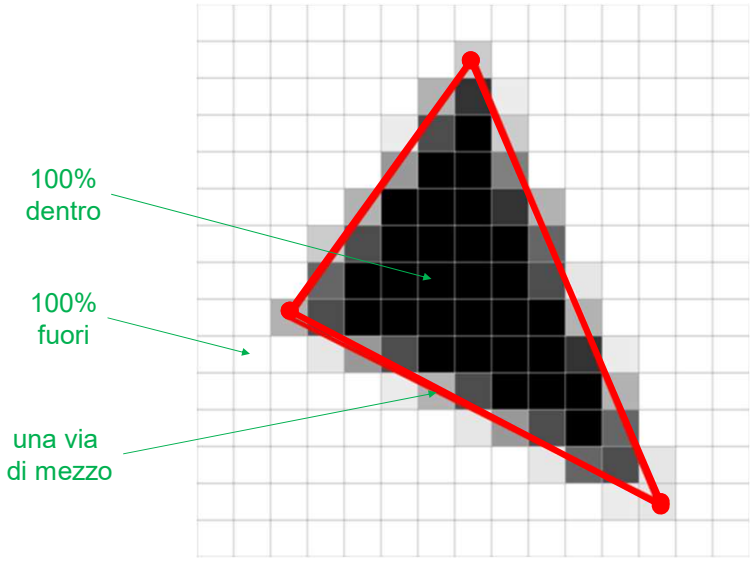
57

Antialiasing

- ✓ Negli esempi visti, il rasterizzatore produce o scarta frammenti “o tutto o niente”
- ✓ L’antialiasing (a volte riferito con la sigla AA) è il nome di un insieme di tecniche per nascondere il difetto di scalettatura indotto dai pixel (aliasing)
- ✓ Esistono degli algoritmi di rasterizzazione che producono frammenti «parziali», cioè semitrasparenti, sui bordi delle primitive

58

Antialiasing: nel rasterizzare triangoli

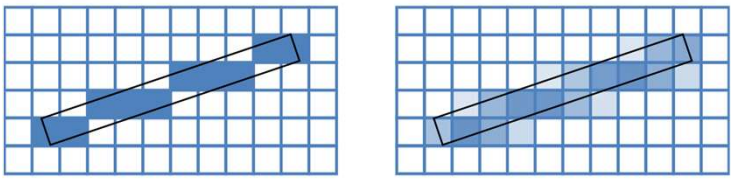


59

Antialiasing: nel rasterizzare segmenti

Senza:

Con:



60