Rendering

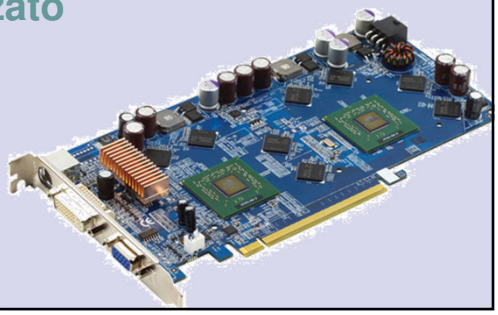
(recall?) Game Engine

- Parte del game che si occupa di alcuni dei task “comuni”
  - Scena / livello
  - Renderer
    - Real time transofrm + lighting
    - Models, materials …
  - Physics engine
    - (soft real-time) newtonian physical simulations
    - Collision detection + response
  - Networking
    - (LAN – es tramite UTP)
  - Sound mixer e “sound-renderer”
  - Gestore unificato HCI devices
  - Main event loop, timers, windows manager…
  - Memory management
  - Artificial intelligence module
    - Soluz dei sotto task comuni AI
  - Supporto alla localizzazione
  - Scripting
  - GUI (HUD)

Rendering

Real Time 3D Rendering

- Task molto oneroso
  - ma, "embarrassingly parallel"
- Ingrediente *base* della soluzione:
  hardware specializzato



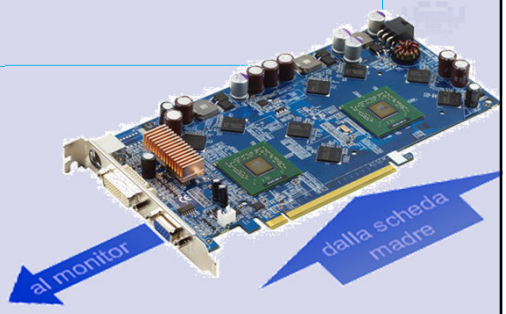
Rendering nei games

- Real time
  - 20 o 30 o 60 FPS
- Hardware based
  - Pipelined, stream processing
- Complessità:
  - Lineare col numero di primitive

Real Time 3D Rendering: API

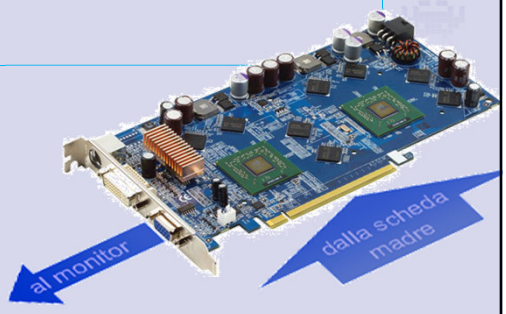
- OpenGL
  - (gruppo Khronos)
- DirectX
  - (Microsoft)

Hardware specializzato per il rendering



- "**GPU**":
  - Graphics Processing Unit
  - La CPU della scheda video
  - *Instruction Set* specializzato!
- Architettura a ***pipeline***
  - a "catena di montaggio"
- Modello di computazione **SIMD**
  - sfrutta l'alto grado di parallelismo insito nel problema
- Possiede la **propria** memoria RAM a bordo
  - "RAM CPU" vs "RAM GPU"
  - grandi copie di memoria da una all'altra dispendiose

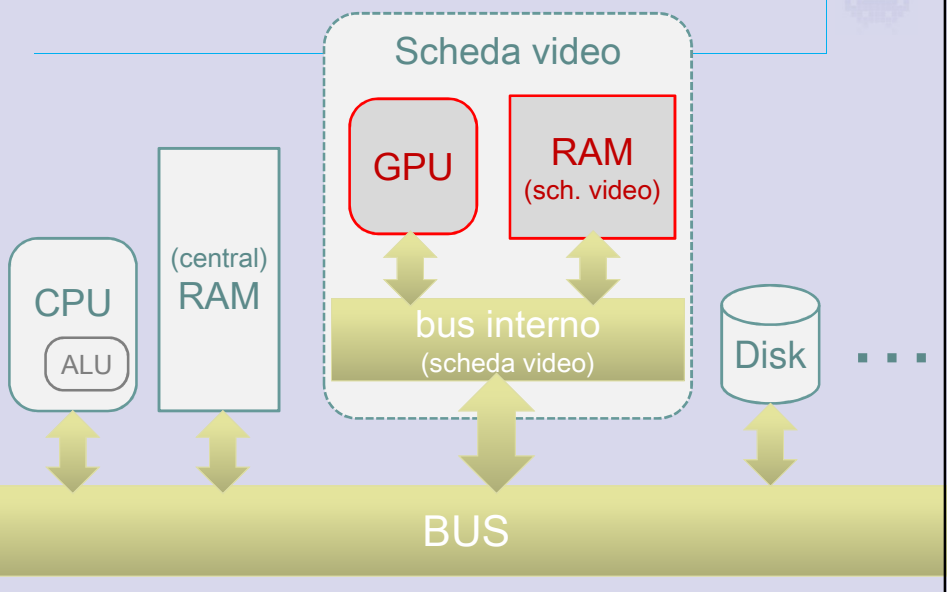
Hardware specializzato per il rendering



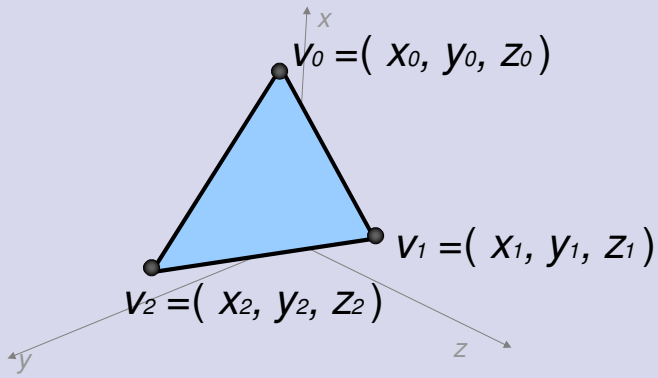
- potenza di calcolo
  - migliaia di GFlops!
- bus molto performante
  - e.g. PCI-express: ~16 GB/s

8

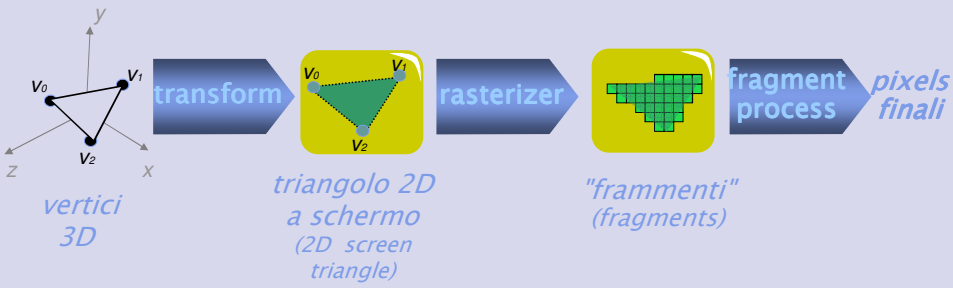
Schema (semplificato)



Il triangolo

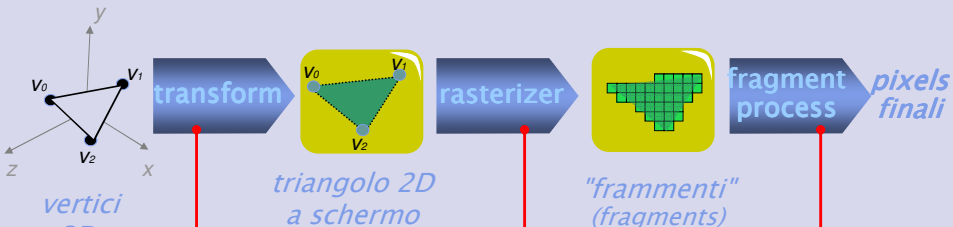


Rasterization-Based Rendering



11

Rasterization-Based Rendering ("transform and lighting")



carico di lavoro per vertice *(sottosistema geometrico)*

carico di lavoro per frammento (~per pixel) *(sottosistema raster)*

12

... Rasterization-Based Rendering

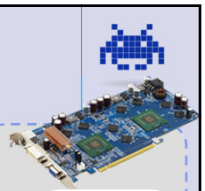
- Dove è il collo di bottiglia?
  - nel sistema geometrico?
    - (l'applicaizone è *transofrm-limited* sinonimo: *geometry-limited*)
  - nel sistema raster?
    - (applicaizone è *fill-limited*)
  - nel bus?
    - (applicaizone è *bus-limited* sin: *bandwidth-limited*)
  - nella CPU?
    - (applicaizone è *CPU-limited*)

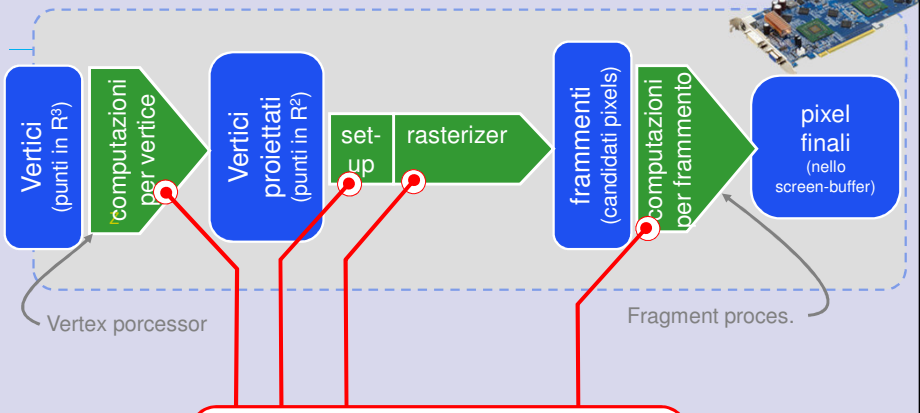
come si può predirre (in teoria)?

come si può verificare in pratica?

perché è importante scoprirlo?

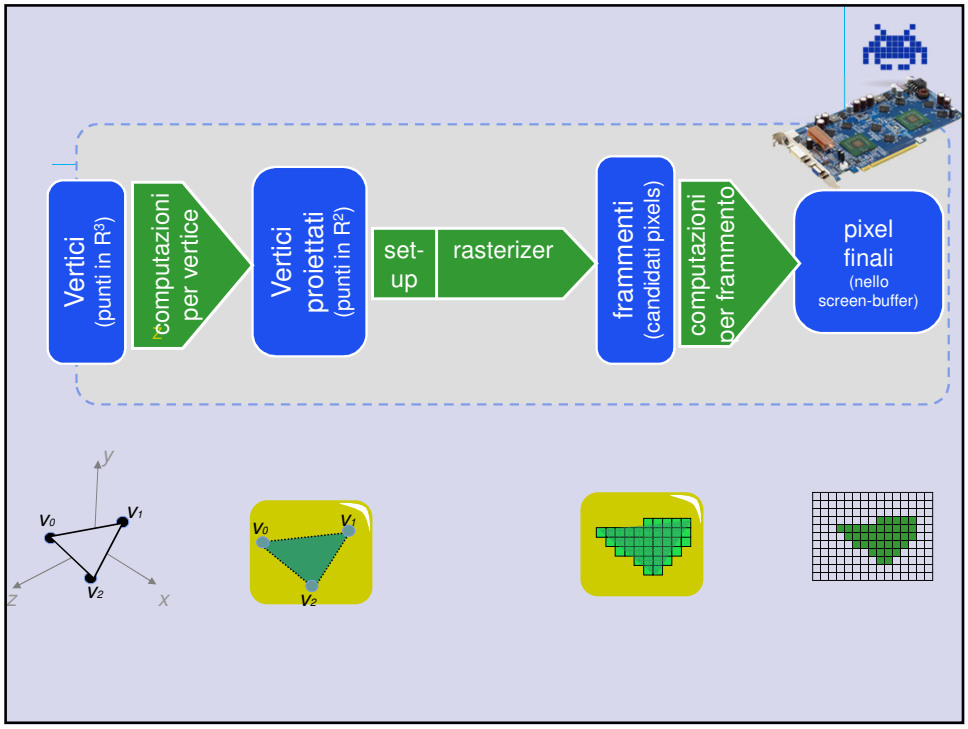
HW support

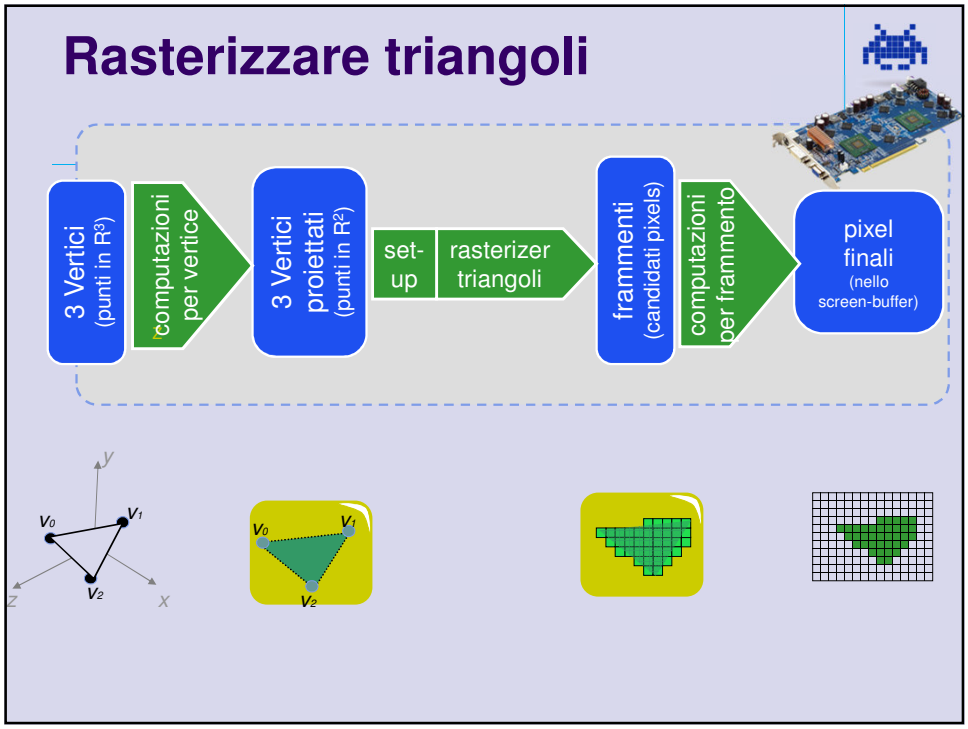


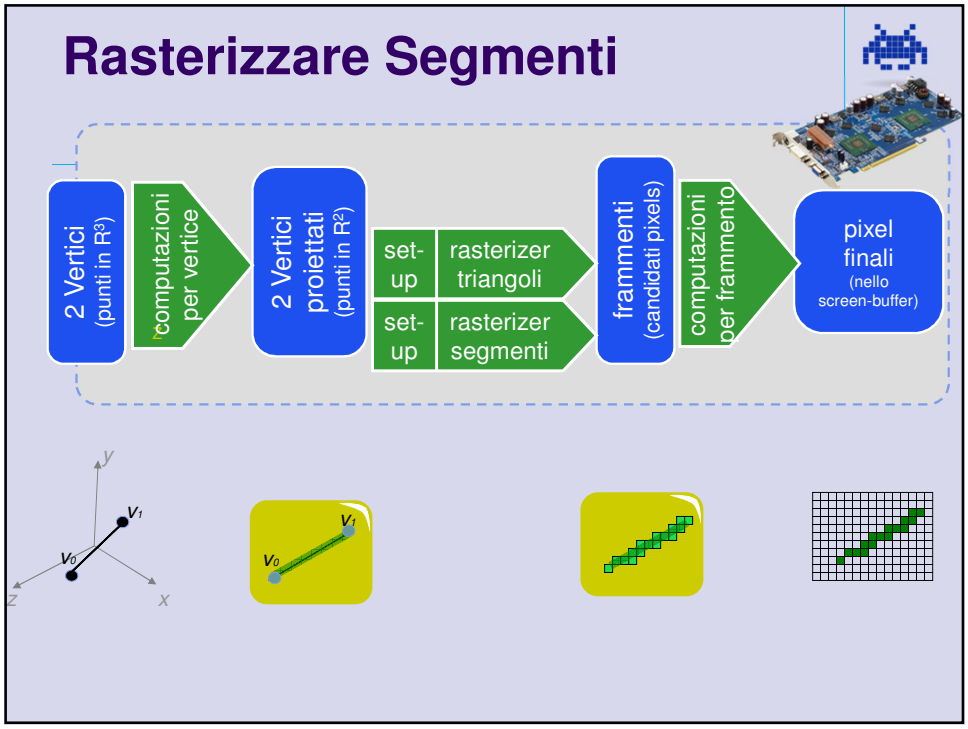


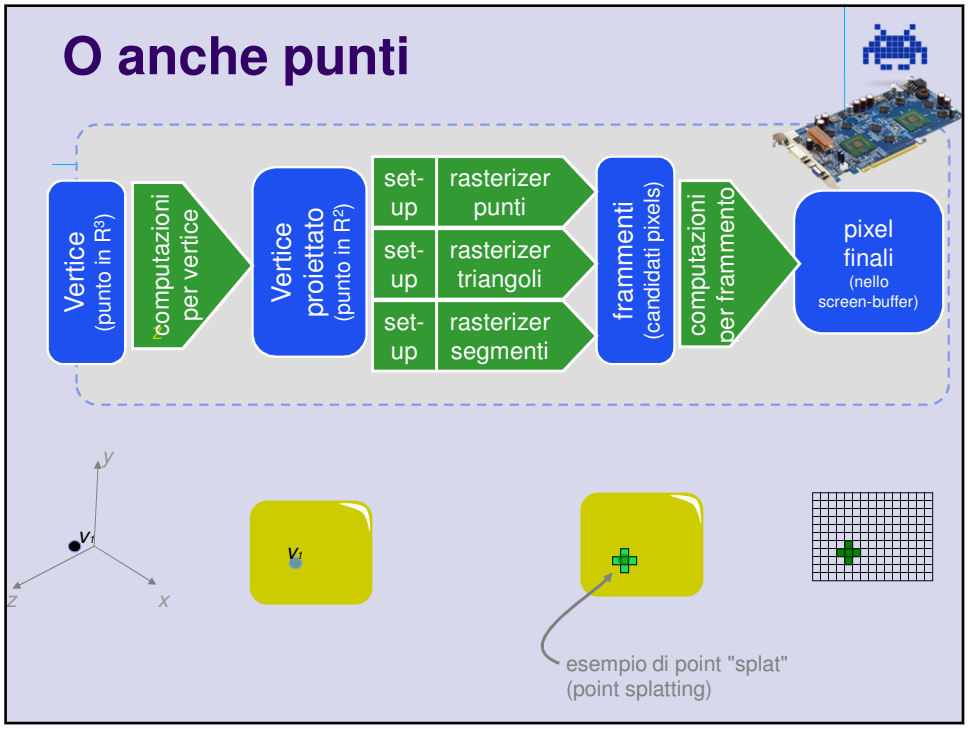
componenti fisiche dell'HW!
Pipeline → Parallelismo → Efficienza

inoltre, molte componenti sono replicate
(negli stages collo di bottiglia)


Vertici
(punti in R³)
computazioni per vertice
Vertici proiettati
(punti in R²)
set-up
rasterizer
frammenti
(candidati pixels)
computazioni per frammento
pixel finali
(nello screen-buffer)
y
x
z
V0
V1
V2


Rasterizzare triangoli
3 Vertici
(punti in R³)
computazioni per vertice
3 Vertici proiettati
(punti in R²)
set-up
rasterizer triangoli
frammenti
(candidati pixels)
computazioni per frammento
pixel finali
(nello screen-buffer)
y
x
z
V0
V1
V2

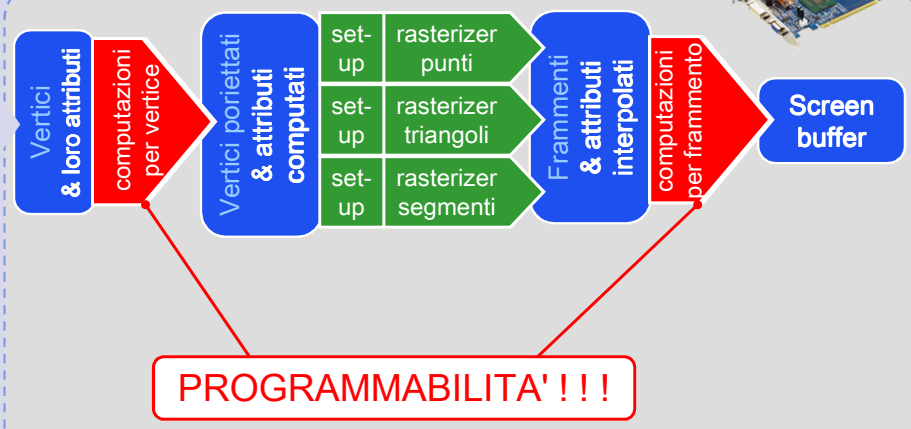

Rasterizzare Segmenti
2 Vertici (punti in R3)
computazioni per vertice
2 Vertici proiettati (punti in R2)
set-up
rasterizer triangoli
set-up
rasterizer segmenti
frammenti (candidati pixels)
computazioni per frammento
pixel finali (nello screen-buffer)
y
x
z
v0
v1

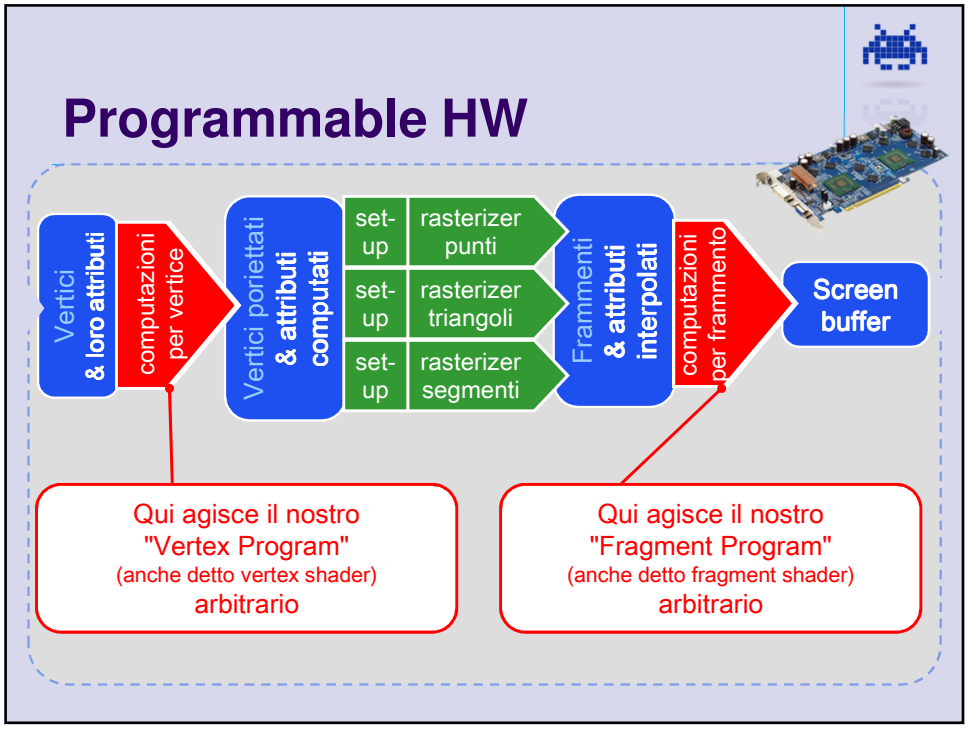

O anche punti
Vertice (punto in R3)
computazioni per vertice
Vertice proiettato (punto in R2)
set-up
rasterizer punti
set-up
rasterizer triangoli
set-up
rasterizer segmenti
frammenti (candidati pixels)
computazioni per frammento
pixel finali (nello screen-buffer)
y
x
z
v1
esempio di point "splat" (point splatting)

in parallelo
(in cascata) (in pipeline)
(a catena di montaggio)

1. Ogni vertice viene ***trasformato***
   - proiettato da spazio 3D (spazio "oggetto") a spazio 2D (spazio "schermo")
   - indipendentemente dagli altri vertici
     - (deve poter avvenire in parallelo!)
   - indipendentemente da quale primitiva fa parte
2. Ogni primitiva viene ***rasterizzata*** in 2D
   - primitiva = triangolo, segmento, o punto
     - rasterizzatore distinti per ogni tipo di primitiva
   - indipendentemente dalle altre primitive
   - rasterizzare = produrre i frammenti corrispondenti
3. Ogni frammento in pos [X,Y] viene processato
   - indipendentemente dagli altri frammenti
   - indipendentemente da quale primitiva lo ha generato
   - output della computazione: un pixel nello screen buffer (RGB)
     - quello a pos [X,Y] (prefissata, la computazione decide solo RGB, non X,Y)

Programmable HW





Rasterization based rendering: schema base

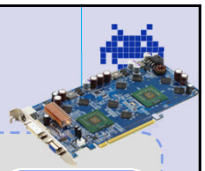
- Per vertice: (vertex shader)
  - transform (da spazio oggetto a spazio schermo)
- Per primitiva: (rasterizer)
  - rasterizzazione
  - interpolazione dati prodotti per vertice
- Per frammento: (fragment shader)
  - lighting (da normale + luci + materiale a RGB)
  - texturing
  - alpha kill
- Per frammento: (dopo il fragment shader)
  - depth test
  - alpha blend

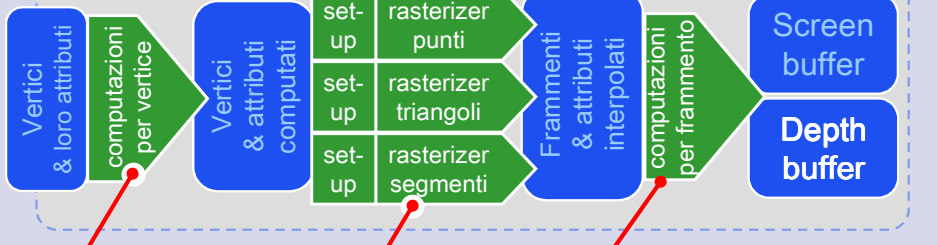
Linguaggi di shading

- Alto livello:
  - **HLSL** (High Level Shader Language, Direct3D, Microsoft)
  - **GLSL** (OpenGL Shading Language)
  - **CG** (C for Graphics, NVidia)
- Basso livello
  - **ARB** Shader Program (come un assembler)

- HLSL e GLSL molto simili
- CG più ad alto livello e pensato per utilizzare sia HLSL che GLSL

Algoritmo dello z-buffer





Transform. Metti la z finale come attributo aggiuntivo

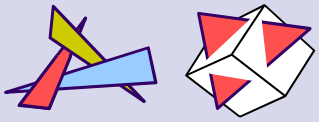
Interpola la z (come tutti gli attributi)

per un frammento con
screen coordinates (x,y),
colore (r,g,b) e profondità z:

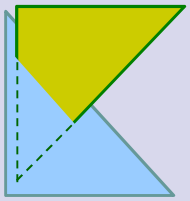
```
if ( z <= DepthBuffer[x,y] ) {   depth test
    ScreenBuffer[x,y] = (r , g , b) ;
    DepthBuffer[x,y] = z ;
}
else scarta (“discard”, “kill”) frammento
```

Algoritmo dello z-buffer: proprietà

- "order independent" ☺!!!
- Molto robusto
  - funziona anche su:

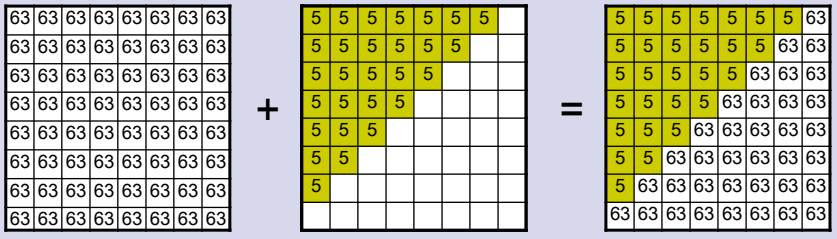


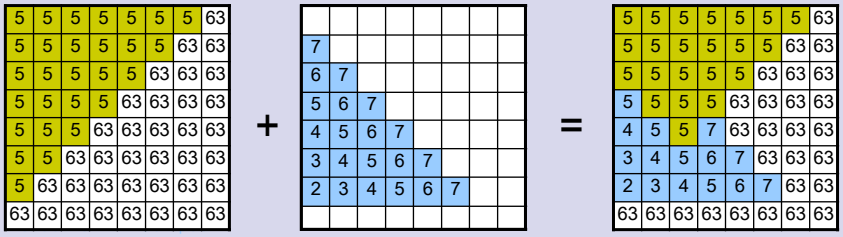
5	5	5	5	5	5	5	63
5	5	5	5	5	5	63	63
5	5	5	5	5	63	63	63
5	5	5	5	63	63	63	63
4	5	5	7	63	63	63	63
3	4	5	6	7	63	63	63
2	3	4	5	6	7	63	63
63	63	63	63	63	63	63	63



- Eseguire un rendering = costruire un depth test (come effetto collaterale)

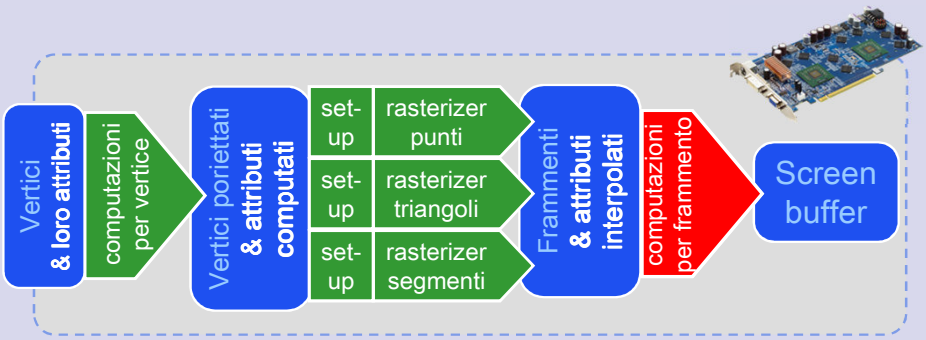
Algoritmo dello z-buffer: esempio a 63)



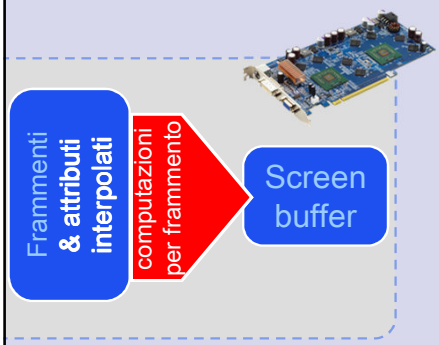


Alpha Blending

- Semitrasparenze



Alpha Blending



- I colori hanno 4 componenti:
  - R,G,B, α
- Dato un frammento
  - (che sopravviva al depth test)

  invece di sovrascriverlo, uso la formula

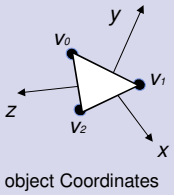
$$(r,g,b)_{finale} = \underbrace{(r,g,b)_{vecchio} \cdot (1-\alpha) + (r,g,b)_{nuovo} \cdot (\alpha)}_{\text{"alpha blending"}}$$

Alpha Blending

Frammenti **& attributi interpolati** → computazioni per frammento → Screen buffer

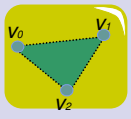
- Il fragment shader dovrà dare in output:
  - …un colore RGB e…
  - …una profondità e…
  - **un parametro alpha**
    - la trasparenza di ***quel*** pixel
- e' la quarta componente del colore RGBα

Parte I: Transform



object Coordinates

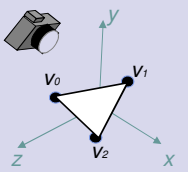
0) transformazione di modellazione
1) transformazione di vista
2) transformazione di proiezione
3) transformazione di viewport



screen Space

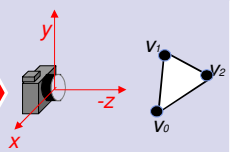
0

3



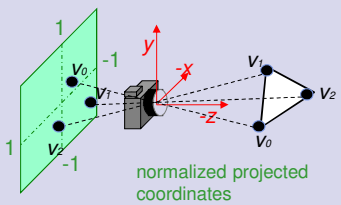
world Coordinates

1



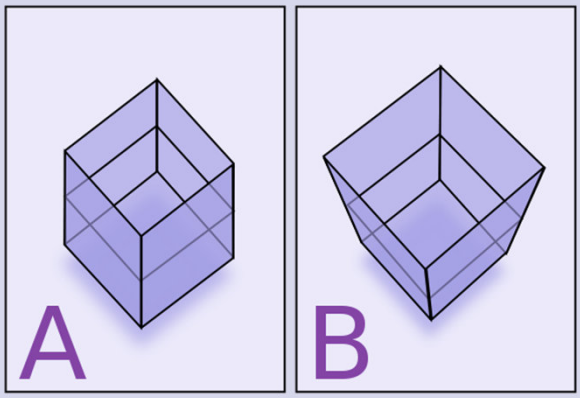
view Coordinates
(a.k.a. eye Coordinates)

2



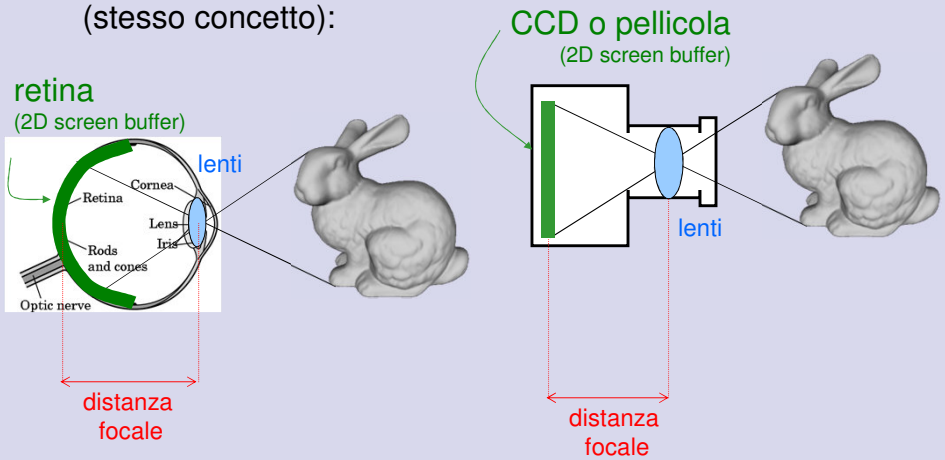
normalized projected coordinates

Trasformazione di proiezione: deformazione prospettica

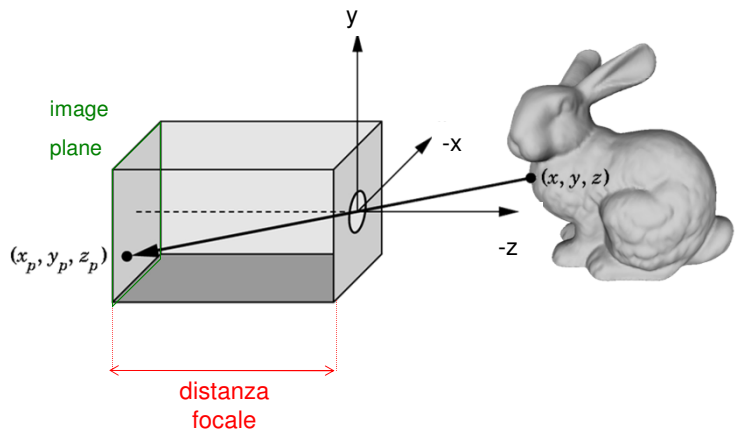


Come si svolge fisicamente il processo:

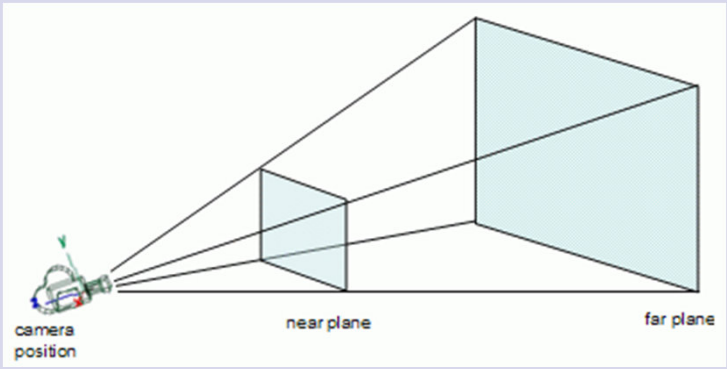
- Occhio o macchina fotografica (stesso concetto):



Pin-hole camera



View Frustum



Pin-hole camera: parametri

- Estrinseci
  - posizione, orientamento
- Intrinseci
  - Lunghezza focale,
    - (equivalentemente, angolo Field of View, FOV)
- Distanze di near plane, far plane

Rendering parte II: lighting base

- Lighting locale
  - (vediamo dettagli prossima lezione)

Rendering parte III: tecniche avanzate diffuse nei games

- Shadowing
  - shadow volumes
  - shadow mapping
  - Screen Space Ambient Occlusion
- Camera lens effects
  - Flares
  - limited Depth Of Field
- Motion blur
- High Dynamic Range
- Non Photorealistic Rendering
  - contours
  - toon BRDF
- Texture based techniques
  - Bumpmapping
  - Parallax mapping

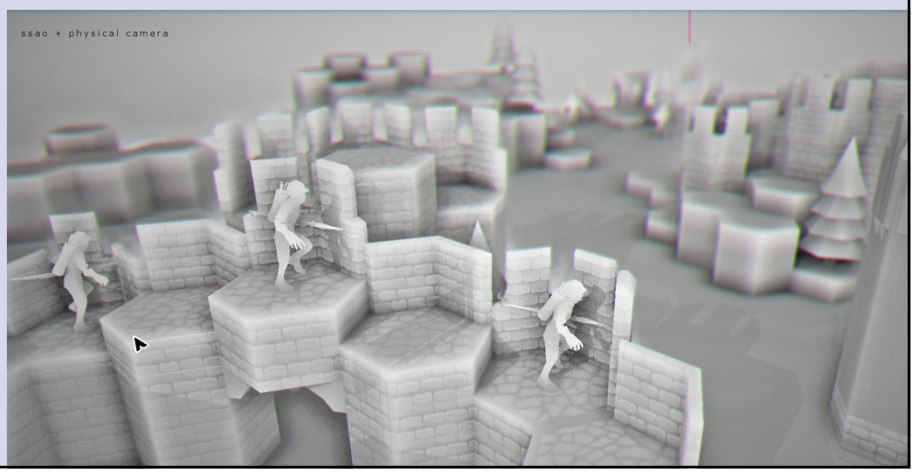
con **PCF**

SSAO

DoF

HDR

NPR



Shadowing: shadow maps

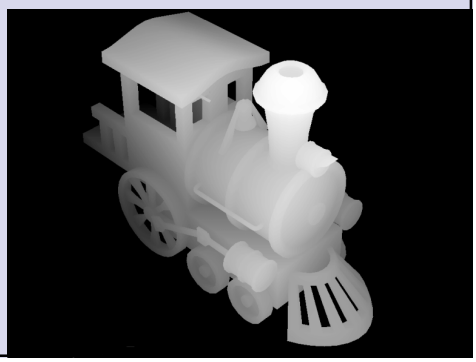
Due renderings:

1: dal punto di vista della luce

- tieni solo il depth buffer

2: della camera

- usa il depth buffer precedente per determinare luce/ombra



Shadow mapping