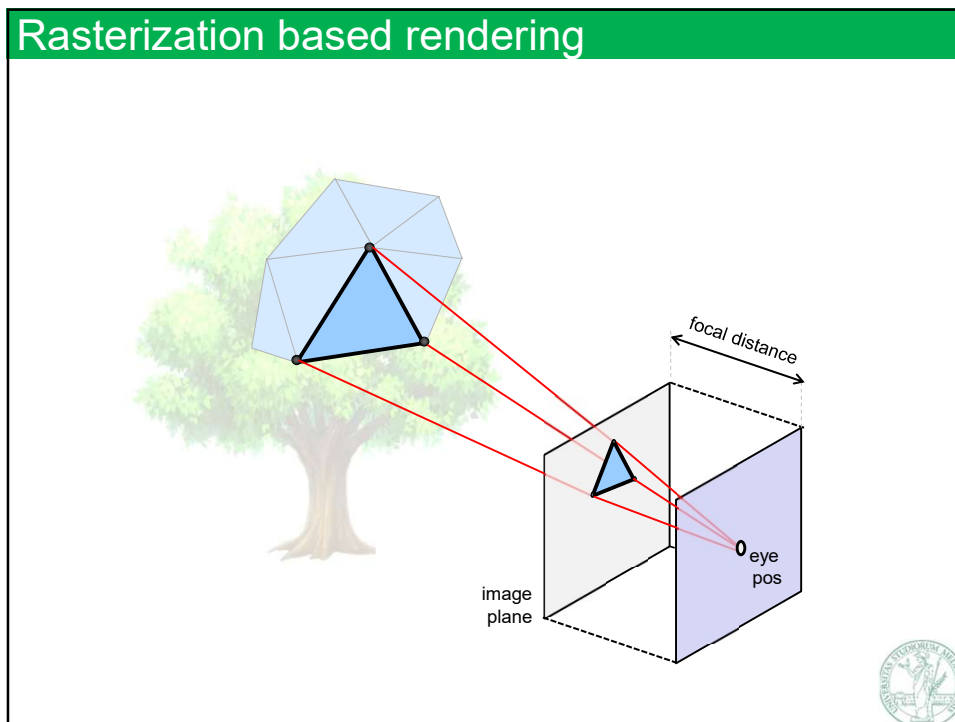


65

Algoritmi di rendering: due categorie

- ✓ Bastati su **Ray-tracing**:
 - ⇒ *per ogni pixel*:
 - lancio un raggio;
 - trovo le intersezioni del raggio con le primitive;
 - determino il colore del punto colpito (per es, calcolando le luci da cui è raggiunto, l'illuminazione...)
- ✓ Basati su **Rasterization**
 - ⇒ *per ogni primitiva*:
 - ⇒ la proietto sullo schermo, in 2D (“trasformazione”)
 - ⇒ converto la forma 2D in pixel (“rasterizzazione”)
 - ⇒ determino il colore di questi pixel

66

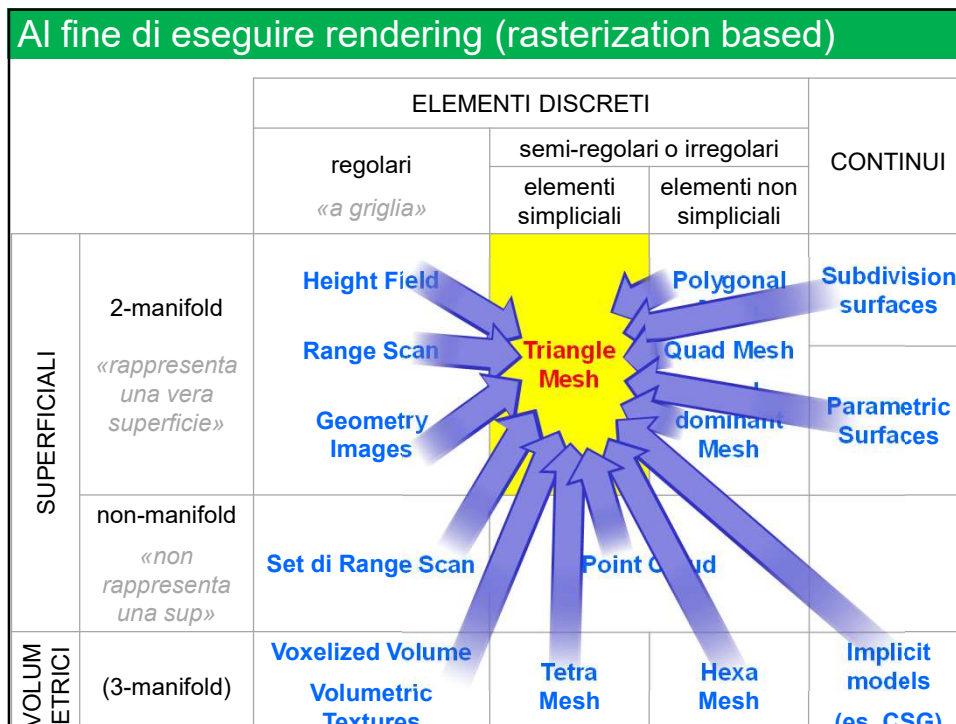


69

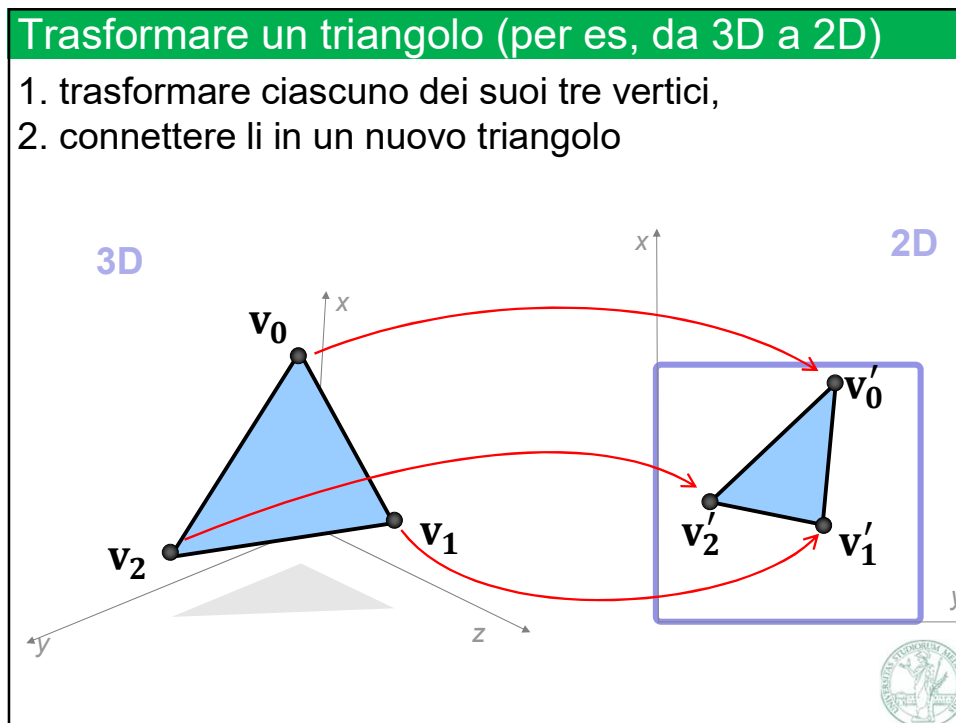
Algoritmi di rendering basati su rasterizzazione

- ✓ Sono resi molto popolari dal supporto di Hardware specializzato: GPU
 - ⇒ è proprio l'algoritmo per cui questo Hardware è stato originalmente pensato e progettato!
 - ⇒ Rendering accelerato con GPU: necessario per rendering in tempo reale (per es, videogame)
 - ⇒ (Recentemente: anche algoritmi basati su ray-tracing vengono spesso accelerati su GPU)
- ✓ La principale (e quasi unica) primitiva di rendering supportata è il triangolo
 - ⇒ Cioè, la GPU è in grado di trasformare e, soprattutto, rasterizzare triangoli
- ✓ Dunque, l'Hardware-supported rendering in tempo reale è soprattutto rendering di Triangular meshes
 - ⇒ E' questo il motivo per il quale, nella prima metà del corso ci siamo occupati così spesso di come convertire qualsiasi altro tipo di modello in una mesh triangolare

70



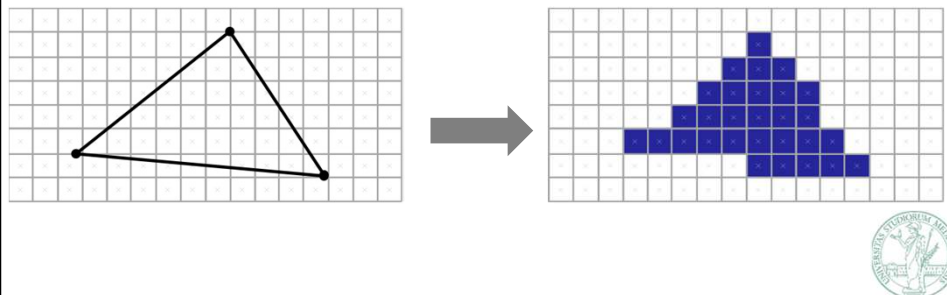
71



73

Rasterizzare (convertire in pixel) un triangolo

- ✓ Trovare quali «frammenti» generare
 - ⇒ uno per ogni pixel dell'immagine contenuto dal triangolo
 - ⇒ «Frammento» = un piccolo pacchetto di dati per il quale vogliamo generare un pixel (contiene esempio: normale, colore di base, materiale...)
 - ⇒ Lo distinguiamo da «pixel»: il colore RGB finale generato per un frammento

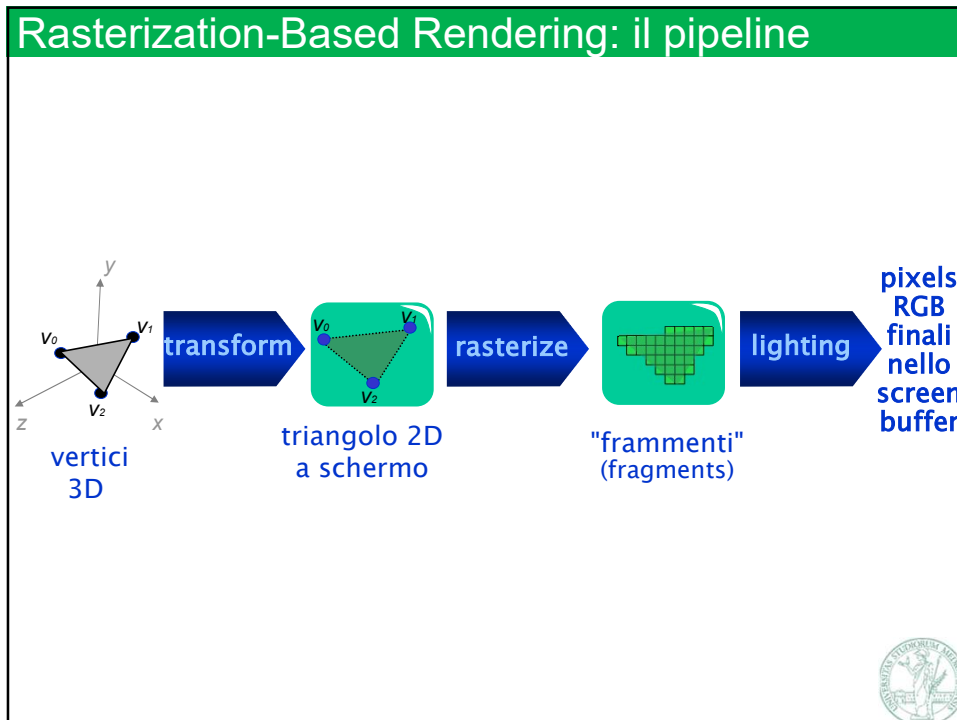


74

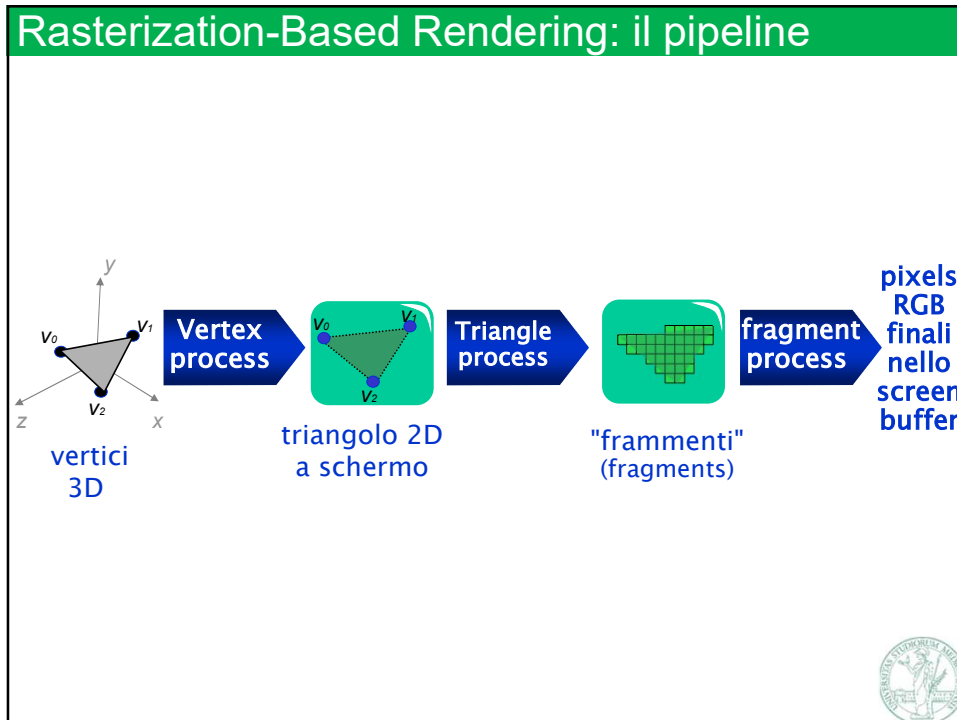
Anche detto T&L: Transform & Lighting

- ✓ *Transform* :
 - ⇒ trasformazioni di sistemi di coordinate
 - ⇒ scopo: portare le primitive in spazio schermo
- ✓ *Lighting* :
 - ⇒ computo illuminazione
 - ⇒ scopo: calcolare il colore RGB finale di ogni pixel della immagine finale

77



78



79

Rasterization-based rendering

- ✓ Input: una mesh (vettore di vertici e triangoli)
- ✓ L'algoritmo a pipeline (catena di montaggio) consiste in diverse fasi:
 1. Fase **per vertice**:
ogni vertice viene portato in una posizione 2D sullo schermo
⇒ Si tratta di una "trasformazione spaziale"
 2. Fase **per triangolo**:
ogni triangolo 2D viene rasterizzato a schermo
⇒ Viene cioè identificato un "frammento" per ogni pixel coperto dal triangolo 2D
 3. Fase **per frammento**
⇒ Per ogni frammento, si computa il colore RGB del pixel corrispondente (tipicamente, calcolando l'illuminazione)

81

Considerazioni generali sul Pipeline

- ✓ Ogni fase del pipeline avviene in parallelo
⇒ Ogni vertice, triangolo, frammento può essere processato in contemporanea con ciascun'altro
- ✓ Le 3 fasi sono in cascata
- ✓ Il pipeline va tanto veloce quanto la sua fase più lenta
⇒ Detta il collo di bottiglia
⇒ Se il collo di bottiglia avviene per vertice, l'applicazione si dice «transform limited»
⇒ Se avviene per frammento, l'applicazione si dice «fill limited»

82

Rendering Algorithms Paradigms

RAY-TRACING

```
For each image pixel  $p$ :  
  make a ray  $r$   
  for each primitive  $o$  in scene:  
    if intersect( $r, o$ )  
    then find color for  $o$   
      color  $p$  with it
```


LIGHTING

RASTERIZATION BASED:

```
For each primitive  $o$ :  
  find where  $o$  falls on screen  
  rasterize 2D shape  
  for each produced pixel  $p$  :  
    find color for  $o$   
    color  $p$  with it
```

PROJECTION 3D \rightarrow 2D (aka TRANSFORM)

LIGHTING



83

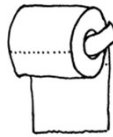
Rendering Algorithms Paradigms

RAY-TRACING


```
for each pixel:  
  for each primitive
```

RASTERIZATION


```
for each primitive:  
  for each pixel
```



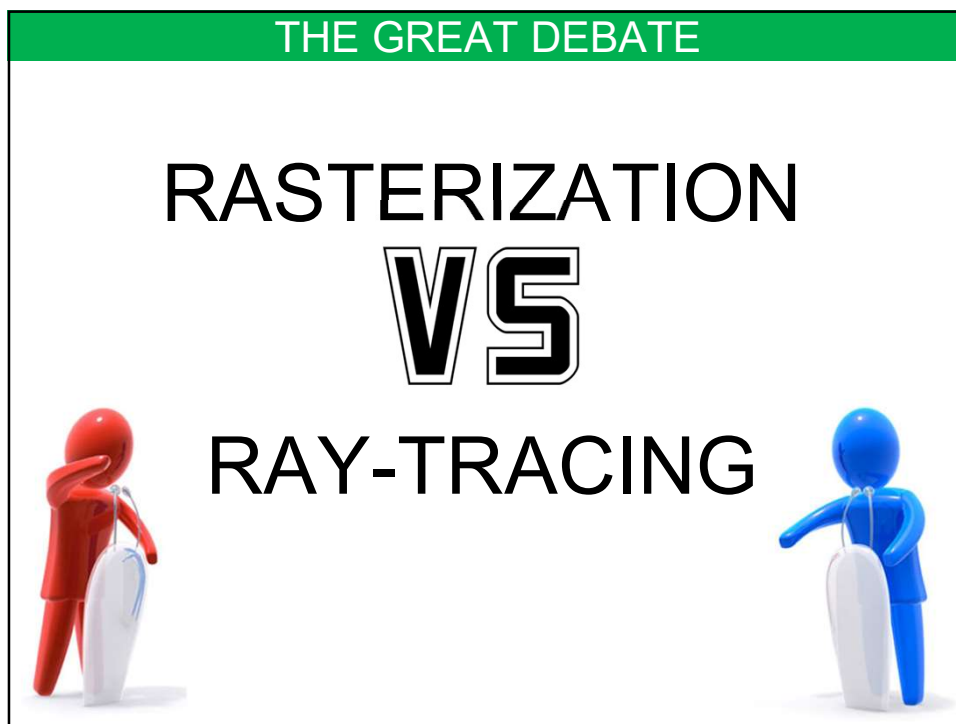
Like this?



Or like this?



84



85


Primitive di rendering

<p>Q: quali primitive di rendering usando rasterization?</p> <p>A: qualunque cosa si sappia:</p> <ol style="list-style-type: none">1) trasformare da 3D a 2D2) «rasterizzare» (in 2D) <p>convertire in pixel</p> <p>per le GPU</p> <p>Quindi, allo stato attuale:</p> <ol style="list-style-type: none">1. PUNTI2. SEGMENTI3. TRIANGOLI <p>per es, point splatting per point clouds</p> <p>Per es, per triangle mesh</p> <p>Il caso principale! (di gran lunga). Le GPU sono ottimizzate per questo caso</p> <p>E' il motivo principale della popolarità delle tri-mesh come prim</p>	<p>Q: quali primitive di rendering usando ray-tracing?</p> <p>A: qualunque cosa si sappia:</p> <ol style="list-style-type: none">1) intersecare con un raggio <p>efficacemente!</p> <p>Per es, per triangle mesh</p> <p>Quindi:</p> <p>⇒ Triangoli?</p> <p>Anche, ma anche primitive più complesse, come:</p> <p>⇒ Implicit surfaces</p> <p>⇒ Height fields</p> <p>⇒ Voxelized dataset</p> <p>⇒ ...</p> <p>Per es, per CSG</p> <p>Il metodo principale per «direct volume rendering»</p>
---	--

86

Vantaggi del ray-tracing

- ✓ E' una classe di algoritmi concettualmente semplice
- ✓ E' facile ottenere effetti visuali complessi (e realistici) (incluso rifrazione, riflessioni, ombre...)
- ✓ Simula in modo abbastanza accurato la sintesi di un'immagine da parte di (ad esempio) una macchina fotografica (pur con molte semplificazioni)
- ✓ E' possibile renderizzare direttamente qualsiasi «primitiva di rendering»: basta implementare la funzione di computo di intersezione con un raggio
- ✓ Parallelismo implicito: ogni pixel può essere implementato in parallelo (modello SIMD -- single instruction, multiple data)




87

Ray-tracing : semplicità ed eleganza?

```
typedef struct{double x,y,z;vec r;vec U,black,amb=.02,.02;.struct sphere{vec cen,color;double rad,kd,ks,kt,kl,lr}*s,*best,sph[]={0,6,.,.5,1,1,1,.,.9,.05,.2,.85,0,1,7,-1,.,8,.,-5,1,.,5,2,1,.,7,3,0,.,05,1,2,1,8,.,-5,1,.,8,.,8,1,.,.5,7,0,0,0,1,2,3,.,-6,1,5,1,.,8,1,7,0,0,0,.,6,1,5,.,-3,.,-2,1,.,5,1,.,1,.,5,.,0,.,0,.,.5,1,5,};}x;double u,b,tmin,sgt(),tan(),double vdot(A,B)vec A,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(A,B)double a;vec A,B;(B.x==a*A.x+B.y==a*A.y+B.z==a*A.z;return B;}vec vunit(A)vec A{return vcomb(1./sgt(vdot(A,A)),A,black);}struct sphere*intersect(F,D)vec F,D;(best=>min=1e30; sph=>while(s-->sph)b=vdot(D,U=vcomb(-1.,F,s->cen)),u=b*b-vdot(U,U)+->rad*>rad,u=u>0?sgt(u):1e31,u=b-uble-77b-ub+u,tmin=u>1e-7?uctmin:best,u;tmin;return best;}vec trace(level,F,D)vec F,D;(double d,eta,r;vec S,color; struct sphere*s,*i;if(!level--)return black;if(s=intersect(F,D));else return amb;color=amb;eta=s->lr;d=-vdot(D,S);vunit(vcomb(-1.,F+vcomb(tmin,D),s->cen));if(d<0)w=vcomb(-1.,S,black),eta=1/eta,d=-d;1; sph=>while(1-->sph)if((e=1->kl*vdot(S,U=vunit(vcomb(-1.,F,1+cen))))>0;4;intersect(F,U)=1)color=vcomb(e,1->color,color);U=s->color;color.x**U.x+color.y**U.y;color.z**U.z;e=1-eta+eta*(1-d*d);return vcomb(s->kt,e*0?trace(level,F,vcomb(eta,D,vcomb(eta*d-sgt(e),U,black)););black,vcomb(s->ks,trace(level,F,vcomb(eta,d,U),vcomb(s->kd,color,vcomb(s->kl,U,black)))));}main(){printf("%d %d\n",32,32);while(yx<32*32)U.x=yx32-32/2,U.z=32/2-yx**32,U.y=32/2/tan(25/114.5915590261),U=wcomb(255,trace(S,black,vunit(U)),black);printf("%d %d %d\n",U);}/*minray*/
```

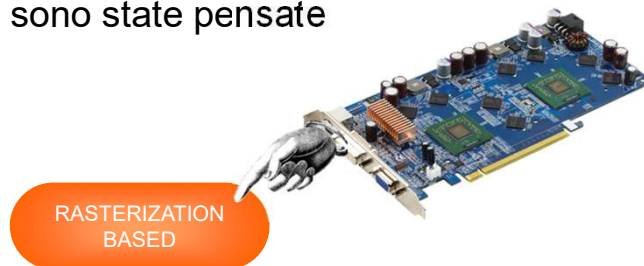
un intero raytracer (in C) su una business card :-P !
(by Paul Heckbert)



88

Vantaggi degli approcci basati su rasterizzazione

- ✓ Complessità lineare con numero di primitive
- ✓ Per ogni primitiva, si processano solo i pixel coinvolti – migliore scalabilità?
- ✓ Attualmente possono contare sul supporto HW su qualsiasi piattaforma
 - ⇒ E' la classe di algoritmo per la quale le GPU sono state pensate



89

Visione storica / tradizionale (il luogo comune)


- ✓ Ray-tracing based
 - ⇒ Lenti, ma ottima qualità di immagini
 - ⇒ Gli algoritmi standard per il rendering offline
 - ⇒ Per esempio, usati nella movie industry
 - ⇒ Alcuni software ray-tracers / path-tracers noti: POV-ray, renderman (pixar), YafaRay, Mitsuba
- ✓ Rasterization based
 - ⇒ Veloce, ma approssimato
 - ⇒ Gli algoritmi standard per il rendering online
 - ⇒ Per esempio, usato nella game industry
 - ⇒ Alcuni API basati su rasterization based: OpenGL, WebGL, DirectX, Metal, Vulkan




90

Render farms

- ✓ Il rendering di entrambi i tipi sono massicciamente parallelizzabili (sono dotati di Implicit parallelism)
- ✓ Per avvantaggiarsene, abbiamo bisogno di hardware parallelo
- ✓ Anche il algoritmi di ray-tracing sono parallelizzato, per es



Pixar/Disney Render Farm



91

Il dibattito dura da decenni

“Real Time Ray-Tracing: The End of Rasterization?” (Jeff Howard, 2007)



Why ray tracing?

Environment map

Ray-traced reflections

PIXAR

93

I luoghi comuni

✓ **Raytracing :**

- ⇒ effetti visuali complessi → realismo
 - ombre, riflessioni speculari, rifrazioni, riflessioni multiple...
- ⇒ *quindi:* perfetto per rendering off-line / hi-quality!
- ⇒ Il rendering della movie industry

✓ **Rasterization:**

- ⇒ veloce
 - per ciascuna primitiva, processa solo i pixel coinvolti (invece di: per ogni pixel, processa tutte le primitive)
- ⇒ *quindi:* perfetto per il rendering real-time e approssimato
- ⇒ Il rendering della game industry



95

La realtà: raytracing non è necessariamente lento

✓ Perché...


- ⇒ È anch'esso intrinsecamente parallelizzabile
 - alto livello di **parallelismo implicito**
 - quindi, implementabile con HW parallelo (per es, render farms)
- ⇒ Le primitive possono essere più varie ed espressive
 - Ciascuna rappresenta una forma più articolata
 - Riduce il numero di primitive necessarie a comporre la scena virtuale
- ⇒ Ma soprattutto: **Ottimizzazioni:** usando apposite strutture dati, è possibile ridurre fortemente il numero di primitive da testare per ogni raggio
 - Strutture di **indicizzazione spaziale**
 - Nota: è un po' più arduo per scene animate
 - Le complessità degli algoritmi di Ray-Tracing può essere resa **sublineare** col numero di primitive



96


Real-time raytracing oggi è comune

Un precursore → (su HW specializzato per questo caso)




Scena: 5 alberi x 1.5 milioni di ▲
28mila girasoli x 35K ▲

OpenRT Project
inTrace Realtime Ray Tracing Technologies GmbH
MPI Informatik, Saarbruecken - Ingo Wald 2004



97


Real-time raytracing oggi è comune



Unity
Real time Raytracing
(2019)



Unreal + Nvidia
Real-time Raytracing
(2019)



100

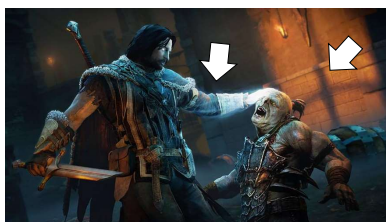
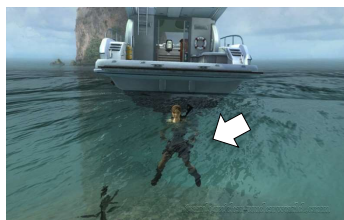
La realtà: rasterization based supporta effetti complessi e realistici

- ✓ Esistono algoritmi basati sul rasterization per simulare, o approssimare:
 - ⇒ Ombre portate
 - ⇒ Diffrazioni
 - ⇒ Riflessioni speculari
 - ⇒ Riflessioni multiple (illuminazione «globale»)
 - ⇒ Rifrazioni e semitrasparenze
 - ⇒ ...e altri effetti tipici e facili negli algoritmi di raytracing
 - ⇒ Persino semplici occlusioni («in una foto, gli oggetti più vicini alla camera nascondono quelli dietro»).
 Rasterization necessita di un'apposita strategia anche solo per questo!
- ✓ Si tratta di algoritmi specializzati, ciascuno pensato per riprodurre un effetto
 - ⇒ Spesso sono adottati nei videogames
 - ⇒ Nota: non sempre combinarli insieme è semplice e diretto



101

La realtà: rasterization based supporta effetti di luce complessi



102

Lo stato attuale dell'industria di intrattenimento

- ✓ L'industria cinematografica si avvale quasi esclusivamente di algoritmi basati su ray-tracing
 - ⇒ Per il rendering finale (offline)
 - ⇒ Ma usa algoritmi basati su rasterization per: effettuare preview (real time), per costruire asset (ad esempio, modellare le mesh), etc
 - ⇒ Rendering finale: parallelizzato massicciamente su render farms, ma cmq spesso lento (ore o anche decine di ore per fotogramma)
- ✓ L'industria dei videogames si avvale quasi esclusivamente di algoritmi basati su rasterization
 - ⇒ Accelerati su GPU consumer level
 - ⇒ Dunque i modelli 3D usati sono sempre tri-meshes
 - ⇒ Trend recente: passare invece ad algoritmi di ray-tracing oppure misti



104

La soluzione del dibattito

“ *Rasterization is fast,
but needs cleverness
to support complex visual effects.*

*Ray tracing supports complex visual effects,
but needs cleverness
to be fast.*

David Luebke (NVIDIA)



106