Marco Tarini - Computer Graphics 2021/2022
Università degli Studi di Milano

Rendering: Rasterization (intro)

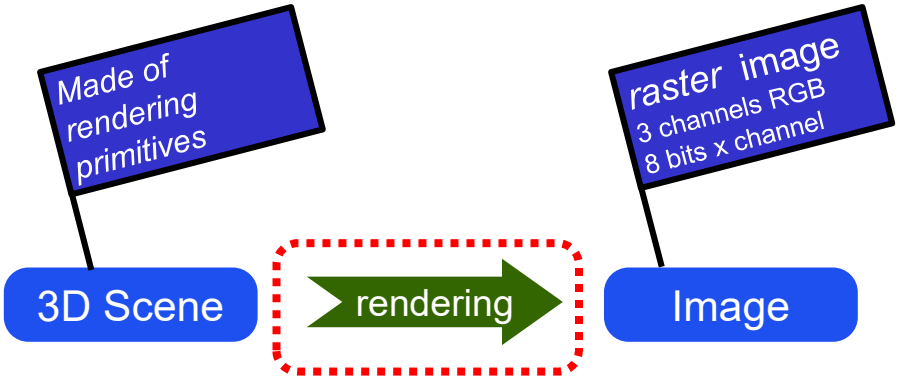
1

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

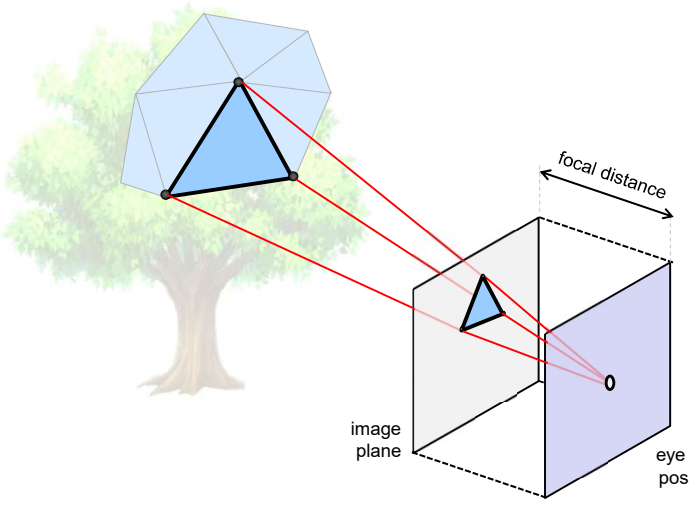
2

PART 2: *the other main approach*



3

Rasterization based rendering



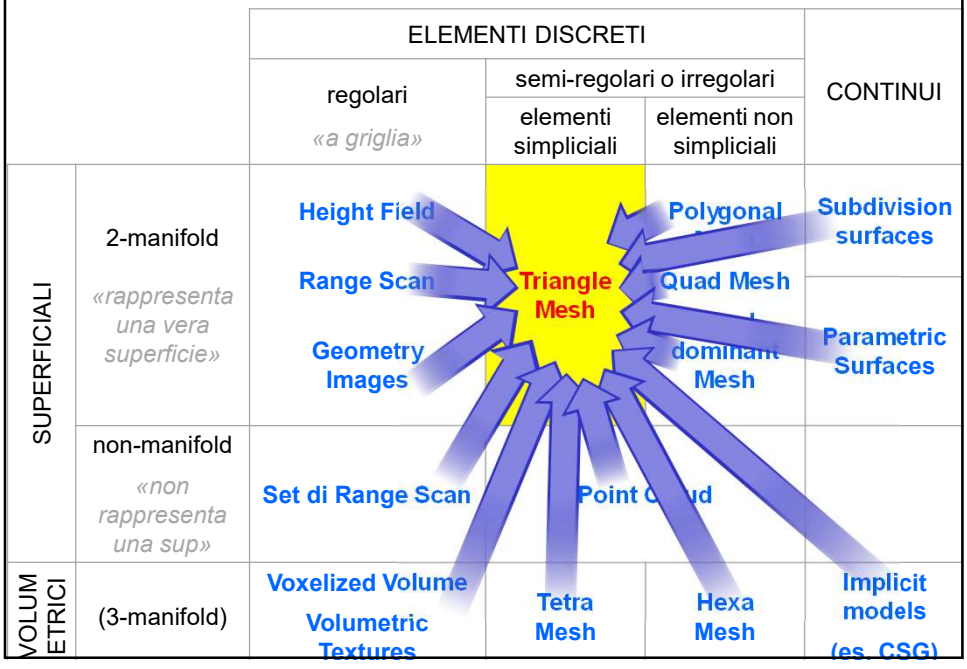
5

Algoritmi di rendering basati su rasterizzazione

- ✓ Sono resi molto popolari dal supporto di Hardware specializzato: GPU
  - ⇒ è l'algoritmo per cui questo Hardware è stato 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, sopratutto, 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

6

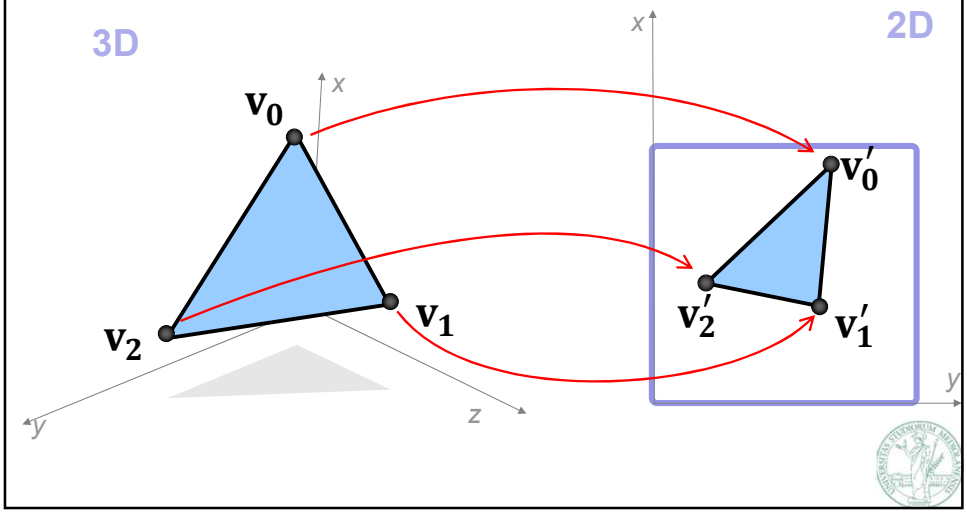
Al fine di eseguire rendering (rasterization based)



7

Trasformare un triangolo (per es, da 3D a 2D)

1. trasformare ciascuno dei suoi tre vertici,
2. connettere li in un nuovo triangolo



9

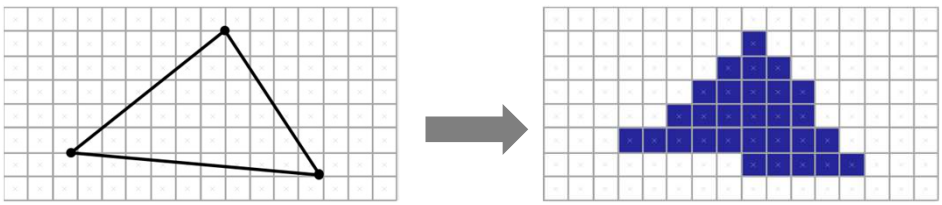
Rasterizzare (convertire in pixel) un triangolo

✓ 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 (esempio: normale, colore di base…)

⇒ Lo distinguiamo da «pixel»: il colore RGB finale generato per un frammento



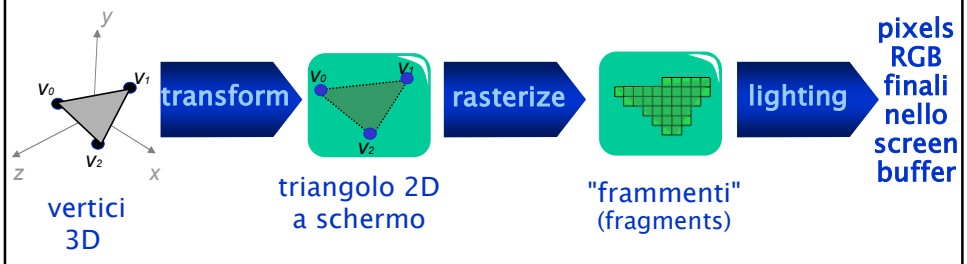
10

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

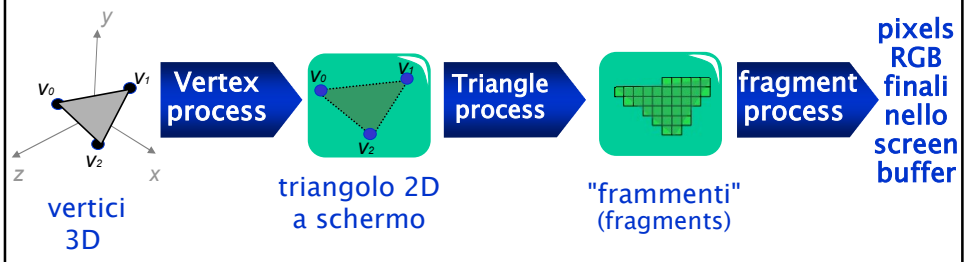
13

Rasterization-Based Rendering: il pipeline



14

Rasterization-Based Rendering: il pipeline



15

Rasterization-based rendering

- ✓ Input: insieme di vertici che formano 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)

17

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»

18

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 → 2D (aka TRANSFORM)

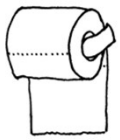
LIGHTING

19

Rendering Algorithms Paradigms

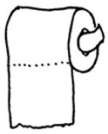
RAY-TRACING

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



RASTERIZATION

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



20

THE GREAT DEBATE

RASTERIZATION

VS

RAY-TRACING

21

Primitive di rendering

Q: quali primitive di rendering usando rasterization?

A: qualunque cosa si sappia:

1) proiettare (da 3D a 2D)
2) «rasterizzare» (in 2D)

convertire in pixel

Quindi, allo stato attuale: per le GPU

1. PUNTI — per es, point splatting per point clouds
2. SEGMENTI
3. **TRIANGOLI** — Per es, per triangle mesh

Il caso principale! (di gran lunga). Le GPU sono ottimizzate per questo caso

E' il motivo principale della popolarità delle tri-mesh come prim

Q: quali primitive di rendering usando ray-tracing?

A: qualunque cosa si sappia:

1) intersecare con un raggio

efficacemente!

Quindi:

⇒ Triangoli? — Per es, per triangle mesh

Anche, ma anche primitive più complesse, come:

⇒ Implicit surfaces — Per es, per CSG
⇒ Height fields
⇒ Voxelized dataset — Il metodo principale per «direct volume rendering»
⇒ …

22

Vantaggi del ray-tracing

- ✓ E' una classe di algoritmi concettualmente semplice
- ✓ E' facile ottenere effetti visuali complessi (e realisitici) (incluso rifrazione, riflessioni, ombre…)
- ✓ Simula in modo abbastanza accurato la sintesi di un'immagine da parte di (ad esempio) una macchina fotografica (pur fra 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)

23

Ray-tracing : semplicità ed eleganza?

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*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.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,}yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,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./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s
->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=1
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->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,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.591559026l),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray!*/
```

un intero raytracer su una buseness card :-P !
(by Paul Heckbert)

24

Vantaggi degli approcci basati su rasterizzazione

- ✓ Complessità lineare con numero di primitive
- ✓ Per ogni primitiva, si processano solo i pixel coinvolti – migliore scalabilità?
- ✓ E' l'algoritmo per il quale le GPU sono state pensate, dunque supporto HW garantito

25

Visione storica / tradizionale (il luogo comune)

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

26

Render farms

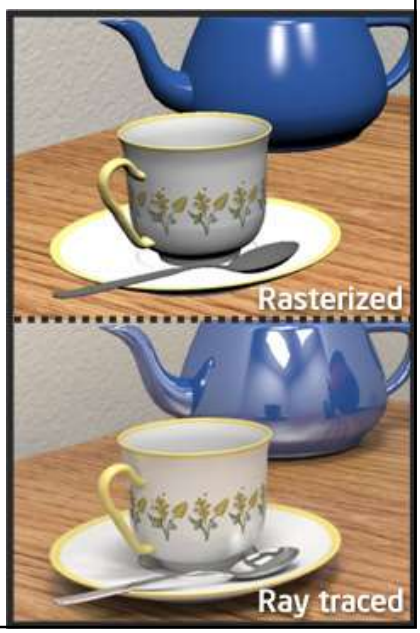
- ✓ Il rendering di entrambi i tipi sono massicciamente paralelizzabili (sono dotati di Implicit parallelism)
- ✓ Per avvantaggiarsene, abbiamo bisogno di hardware parallelo
- ✓ Anche il algoritmi di ray-tracing sono parallelizzato, per es

Pixar/Disney Render Farm

27

Il dibattito dura da decenni

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



29

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

31

La realtà: raytracing non è necessariamente lento

✓ perché:

⇒ È intrinsecamente parallelizzabile
- alto livello di **parallelismo implicito**
- quindi, implementabile con HW parallelo (per es, render farms)

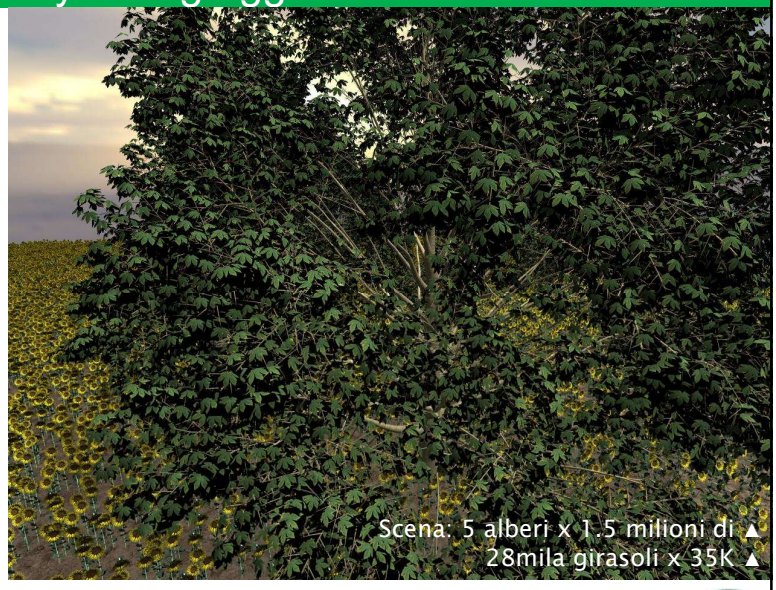
⇒ Le primitive possono essere forme più complesse
- Questo riduce il numero di primitive necessarie a comporre una data scena virtuale

⇒ Ottimizzazioni: usando apposite strutture dati, è possibile ridurre fortemente il numero di primitive da testare per ogni raggio
- Strutture di **indicizzazione spaziale**
- Nota: è più arduo per scene animate
- Le complessità degli algoritmi di Ray-Tracing può essere resa **sublineare** col numero di primitive

32

Real-time raytracing oggi è comune

Un precursore → (su HW specializzato per questo caso)



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

33

Real-time raytracing oggi è comune

Unity
Real time Raytracing
(2019)

Unreal + Nvidia
Real-time Raytracing
(2019)

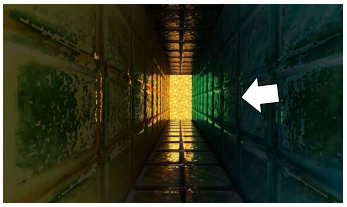
36

La realtà: rasterization based supporta effetti di luce compessi e ralistici

- ✓Esistono molti algoritmi basati sul rasterization per simulare, o approssimare:
  - ⇒Ombre portate
  - ⇒Diffrazioni
  - ⇒Rifelssioni speculari
  - ⇒Riflessioni multiple (illuminazione «globale»)
  - ⇒Rifrazioni
  - ⇒E altri effetti tradizionalmente presenti solo negli algoritmi di raytracing
- ✓Si tratta di algoritmi specializzati, per ciascuno di questi effetti, usati per es nei videogames

37

La realtà: rasterization based supporta effetti di luce compessi



38

Lo stato 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 rastrization
  - ⇒ 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

40

API per rendering basato su rasterizzazione

Alcune API che consentono rendering basato su rasterizzazione, supportata da GPU

- ✓ OpenGL
  (cross platform, cross language)
- ✓ WebGL
  (per web, JavaScript, su browser)
- ✓ GLES
  (per embedded devices, e applicazioni su mobile)
- ✓ DirectX
  (microsoft, quindi inclusa xBox)

41

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)

42