Graphics Programming

Master Computer Game Development 2013/2014

Introduzione alla pipeline di rendering.

University of Verona

luca.giona@3dflow.net

Informazioni generali sul corso

- Contatti:
  - luca.giona@3dflow.net
  - Skype: luca.giona
- Corso teorico (graphics) /pratico (programming)
- Prerequisiti : nozioni basilari di Computer Graphics , algebra lineare e C/C++.
- Ringraziamento a Roberto Toldo (3Dflow) per aver fornito le prime versioni di queste slide.

Informazioni generali sul corso

- Libri di testo seguiti:
  - Teoria :
    - **Real-Time Rendering** *third edition - Thomas Akenine-Moller, Eric Haines and Naty Offman* http://www.realtimerendering.com/
    - **Introduction to 3D Game Programming with DirectX 11** - Frank Luna

    Consigliati ma non obbligatori
  - Pratica :
    - **Visual Studio 2012:** http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-desktop

Un po' di storia: librerie CG per PC

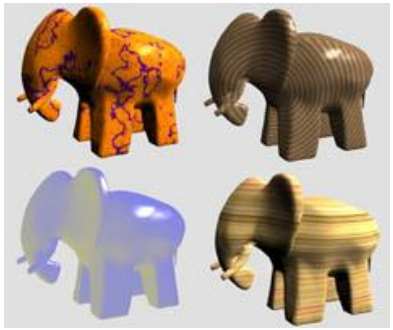

●SGI 初のハードウエア「IRIS 1000」グラフィックスターミナル

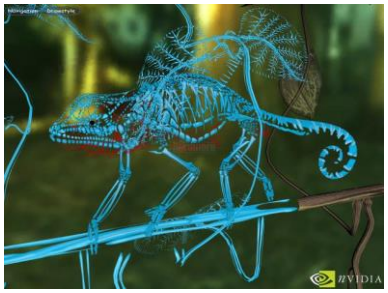
1991 – Opengl 1.0

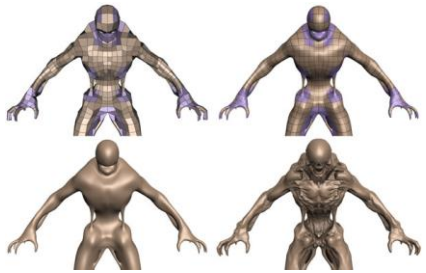
1995 – First consumers 2d/3d video card – Directx 1.0

1997 – 3dfx Voodoo (Mipmapping, Zbuffering, Antialiasing)

Un po' di storia: librerie CG per PC







1999 – Nvidia Geforce 256 Transform and lightning

2002– DirectX 9.0

Shaders (HLSL), OpenGl 1.5 (GLSL)

2007– DirectX 10 – Shader Model 4.0

DirectX 11……

Passato

Elite (1984)

Duke Nukem 3D (1995)

Quake (1996)

Quake 3 Arena (1999)

Presente / Futuro

Grafica 3D Realistica

Grafica 3D «particolare»

Grafica Basilare

Grafica 3d Realistica
su dispositivi mobili

Librerie grafiche (PC)

- OpenGL:
  - Pro: multipiattaforma, facile da imparare, standard "aperto", versioni retrocompatibili.
  - Contro: non è molto usata nella game industry (eccezioni: OpenGL ES per dispositivi mobili. Ps3 e Wii api "*simile*"-OpenGL ES).
- DirectX:
  - Pro: molto usata nella game industry e supportata dai produttori hardware.
  - Contro: standard proprietario (Microsoft) solo per Windows e Xbox, più difficile da imparare rispetto a OpenGL.

Linguaggi Shader

- GLSL (OpenGL Shading Language):
  - Per applicazioni OpenGL.
- HLSL (High Level Shading Language):
  - Per applicazioni DirectX
- Nvidia Cg
  - Compila Shader per applicazioni OpenGL e DirectX
- Tutti sono basati sui costrutti del linguaggio C.
- Sono molto simili tra loro ed offrono le stesse potenzialità. In particolare HLSL e Cg hanno una sintassi quasi equivalente.

In questo corso…

- Obiettivi:
  - In questo corso impareremo le basi della programmazione grafica. (teoria)
  - Vedremo assieme come implementare quanto appreso utilizzando le librerie grafiche DirectX 11 con HLSL e shader Model 4.0. (pratica)
- Cosa non vedremo:
  - Molti effetti e tecniche avanzate ma:
    - Alla fine del corso dovreste avere tutte le basi per approfondire in modo autonomo effetti/tecniche avanzate.

La pipeline di rendering

- La pipeline è responsabile della creazione dell'immagine bidimensionale su schermo a partire dalla scena tridimensionale.
- Tale processo deve essere effettuato in tempo reale.
- Concettualmente divisa in tre stadi:
  - Application (~su CPU)
  - Geometry
  - Rasterizer

Cosa intendiamo per Real Time?

- La velocità con cui le immagini vengono disegnate è importante per l'interattività.
  - Si misura in Frames Per Second (fps) o Hertz (Hz)
- Limite distinguibile occhio umano: ~72 fps
- Di solito vi è sincronizzazione tra refresh rate dello schermo e fps (Vsync):
  - Il programma aspetta "disegna" il nuovo frame solo quando il monitor è pronto per aggiornare l'immagine sullo schermo.
  - Es. Monitor a 60 Hz, programma produce 24 fps -> Frame effettivi 20 (60 / 3).

No Vsync?

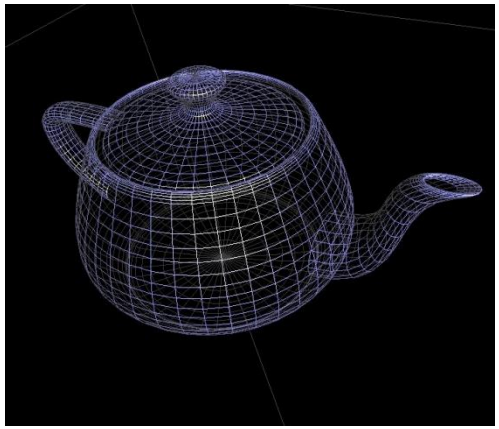
- Se non sincronizziamo monitor e gpu, gli hardware opereranno separatamente.
  - Problema Tearing: l'aggiornamento di un'immagine sullo schermo può avvenire esattamente quando stiamo aggiornando il frame buffer della gpu.
  - Vengono disegnati due "pezzi di frame" diversi a schermo

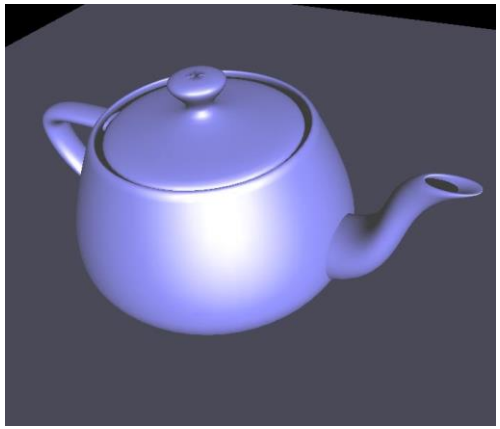
Cosa intendiamo per scena 3D?

- Una scena 3D è composta da:
  - Modelli geometrici
  - Luci
  - Materiali (proprietà dei modelli geometrici) + Texture
- Per proiettare la scena abbiamo bisogno di una telecamera virtuale.
  - Decide dove guardare e come.

Primitive di rendering

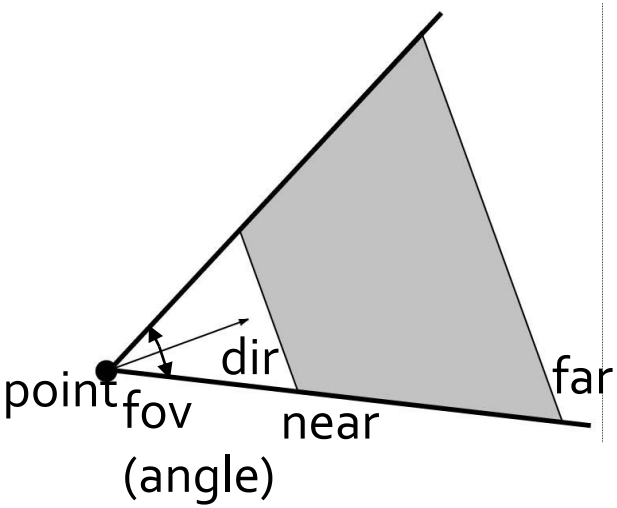
- Utilizzo della scheda video per il rendering real time.
- Le primitive disegnate possono essere solo punti, linee o triangoli (quad e poligoni convessi coplanari scomponibili in triangoli)
- Una superficie è descritta (approssimata) con queste primitive.





Telecamera virtuale

- Definita da:
  - posizione (dove siamo),
  - direzione (cosa guardiamo),
  - vettore Up (com'è ruotata),
  - campo di vista (apertura),
  - piano vicino e lontano (quale porzione dello spazio riusciamo ad osservare)



Application Stage

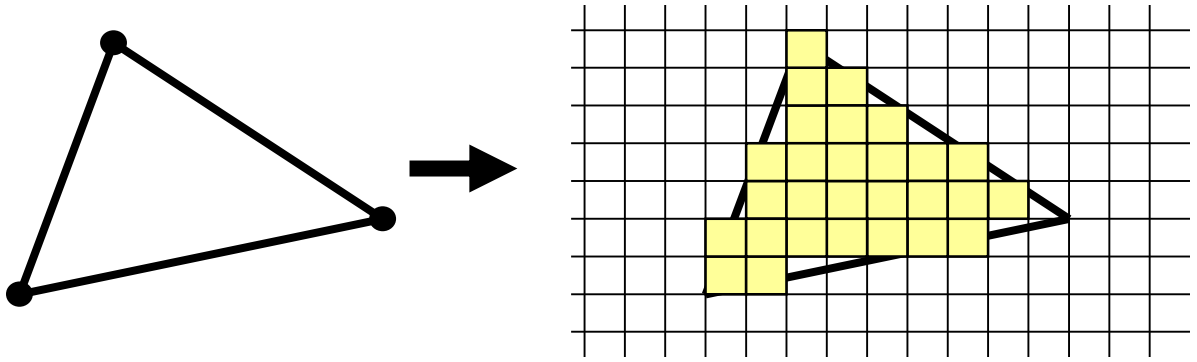
- Eseguita interamente sulla CPU
  - Il programmatore ha pieno controllo
  - Ad esempio utilizzata per Animazione o Collision Detection.
- L'operazione più importante – dal nostro punto di vista - è quella di spedire le primitive di rendering all'hardware grafico.

Geometry Stage

- Obiettivo: svolge operazioni "geometriche" sulla scena
- Ad esempio:
  - Posiziona gli oggetti (matrix multiplication)
  - Posiziona la telecamera (matrix multiplication)
  - Calcola illuminazione modelli (per vertice)
  - Proietta scena da 3d a 2d.
  - Effettua il clipping.
  - Mappa su finestra.

Rasterizer Stage

- Obiettivo: prendere l'output del Geometry Stage e convertirlo in pixel visibili sullo schermo.



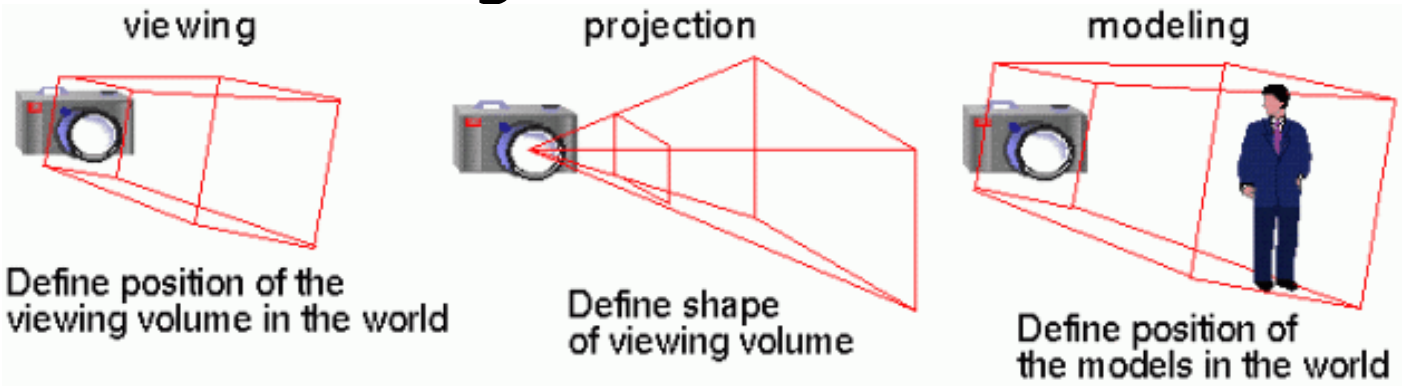
- Aggiunge anche le texture e effettua altre operazioni per-pixel.
- Utilizza lo z-buffer per nascodere oggetti occlusi.

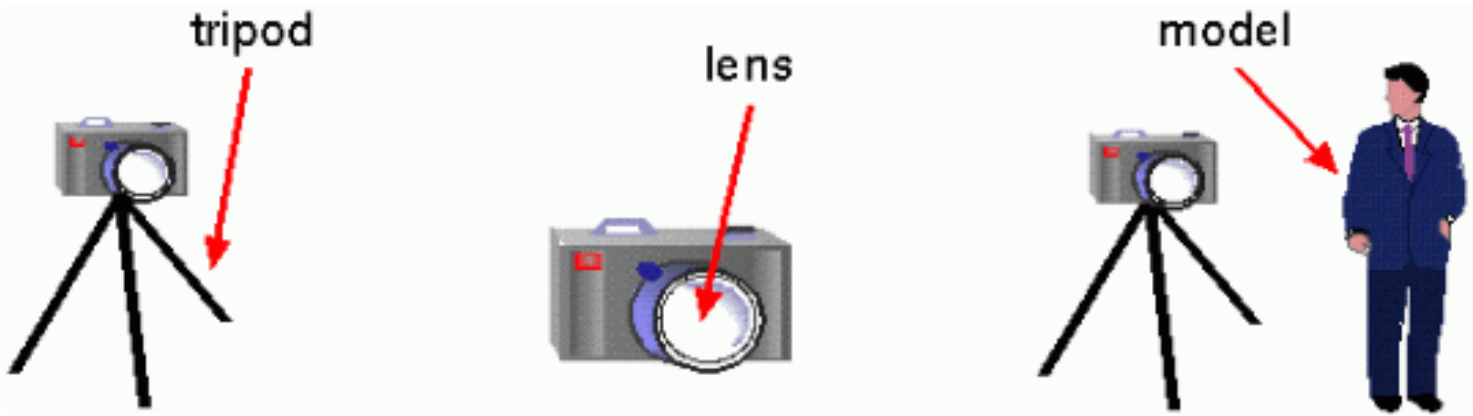
Ricapitolando…

- Il programma definisce le primitive gestite "renderizzate" dalla pipeline (attraverso le API). (Application Stage)
- Il Geometry Stage effettua operazioni sui vertici.
- Il Rasterizer Stage effettua operazioni sui pixel.

Geometry Stage più in dettaglio

- Tre tipi di trasformazioni – Analogia con una macchina fotografica.



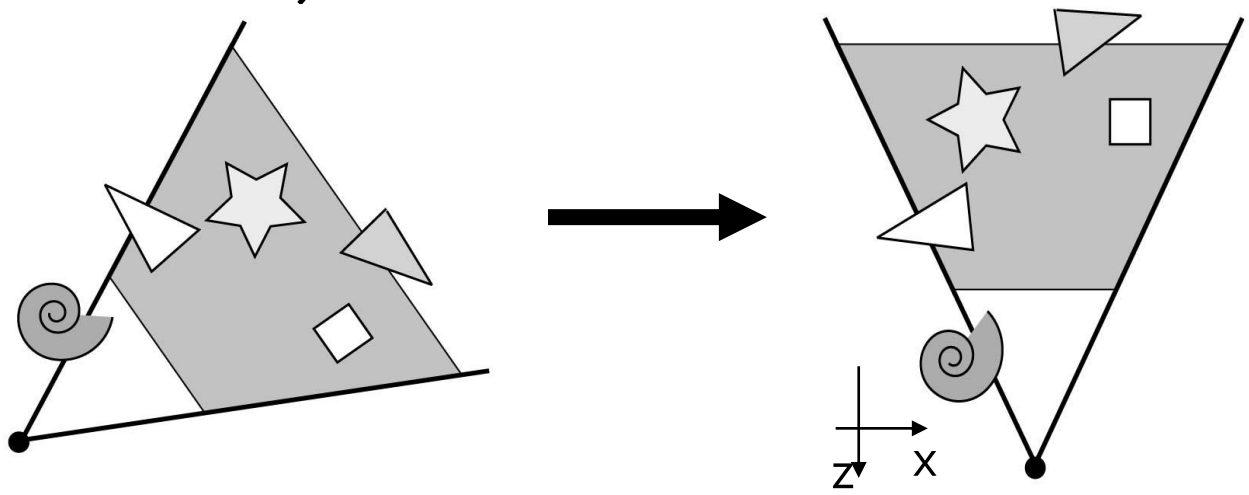


Modeling and Viewing

- Le operazioni di Modeling e di Viewing sono effettuate da matrici 4x4.
- Il modello arriva in coordinate modello locali.
- Viene traslato , ruotato e scalato in "coordinate mondo" (trasformazioni di modeling).
- Successivamente viene applicata a tutti i modelli la view transformation in modo da portare la telecamera virtuale nel punto (0,0,0) che guarda in direzione z positivo (negativa in OpenGL).

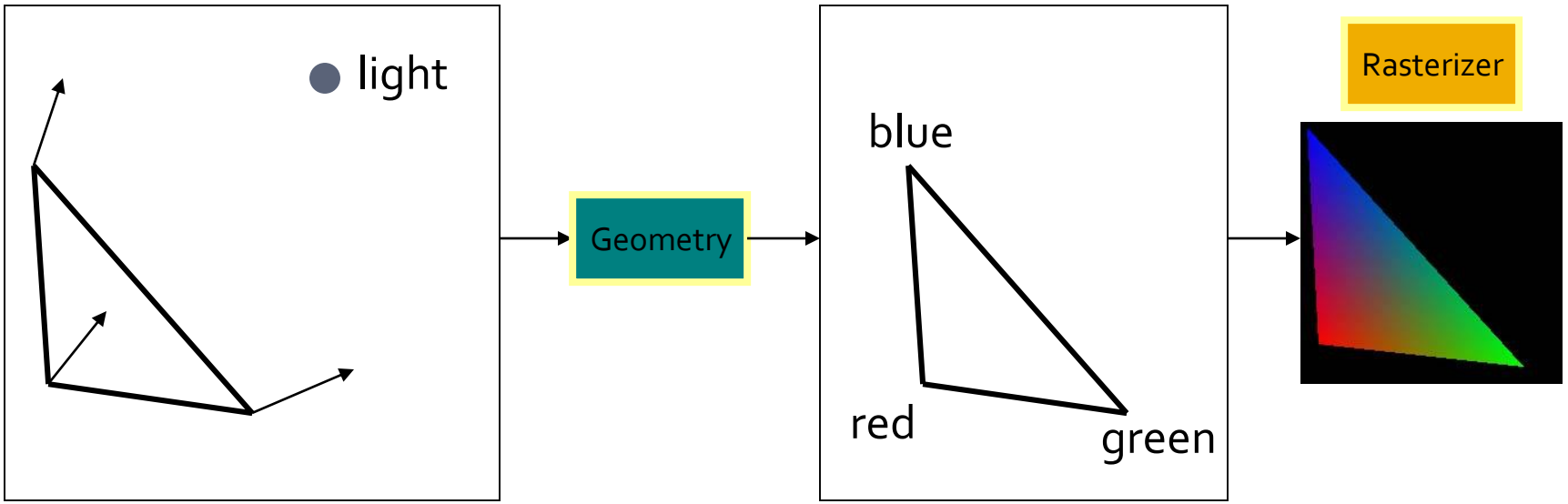
Modeling and Viewing

- Dopo aver applicato le trasformazioni di modeling e viewing, gli oggetti si trovano nello spazio "camera" o "eye"
- Modeling e Viewing possono venire condensate in un'unica matrice 4x4 (modelview)



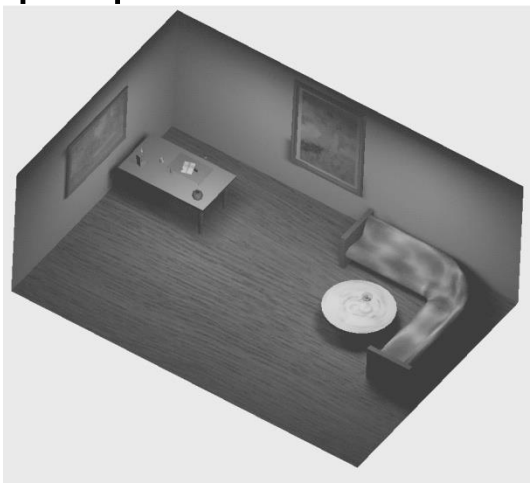
Illuminazione

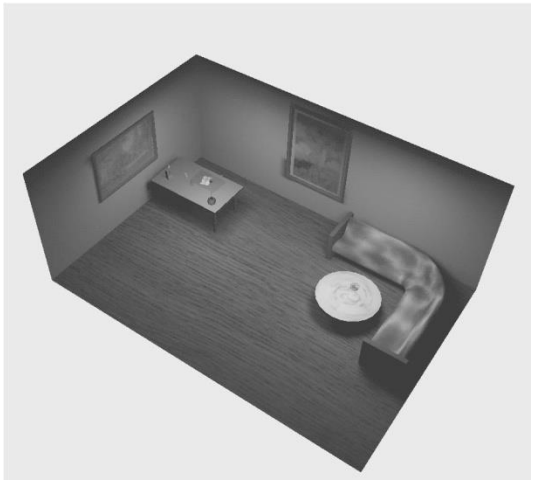
- Calcola l'illuminazione di ogni vertice



Projection

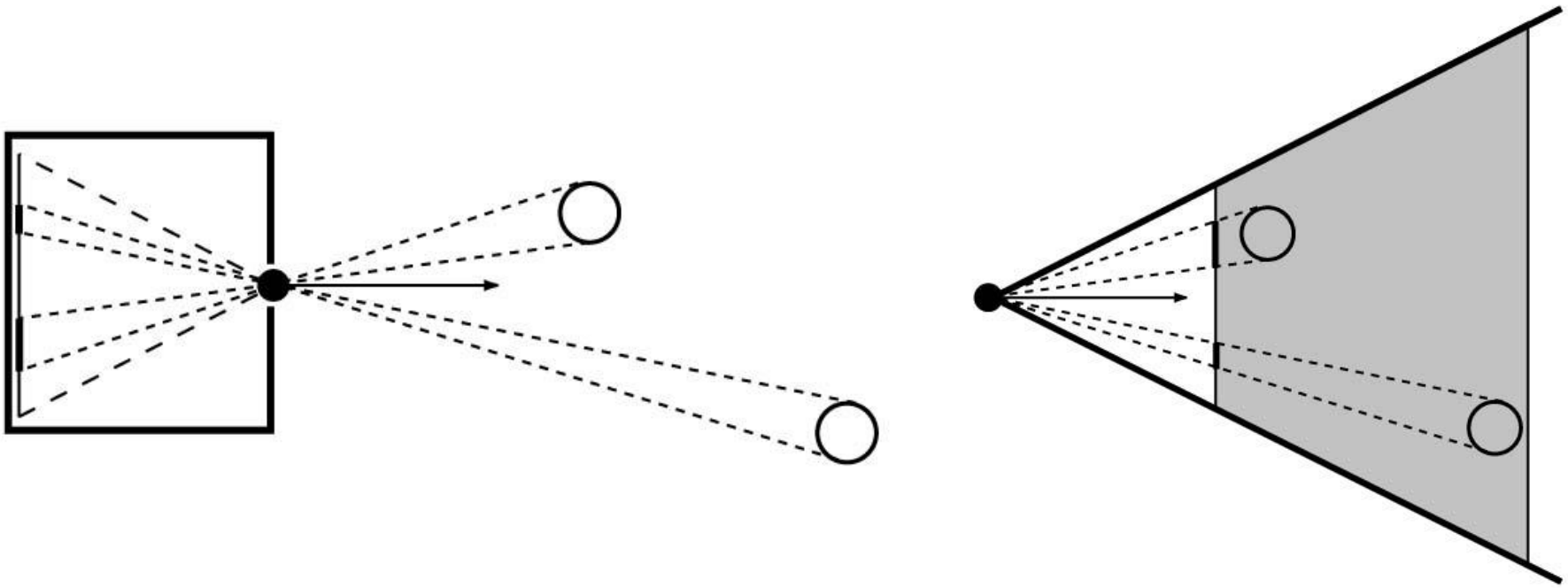
- Due tipi principali:
  - Ortogonale (utile per poche applicazioni, di solito 2d)
  - Prospettica
    - forti indizi tridimensionali – gli oggetti distanti appaiono più piccoli





Projection

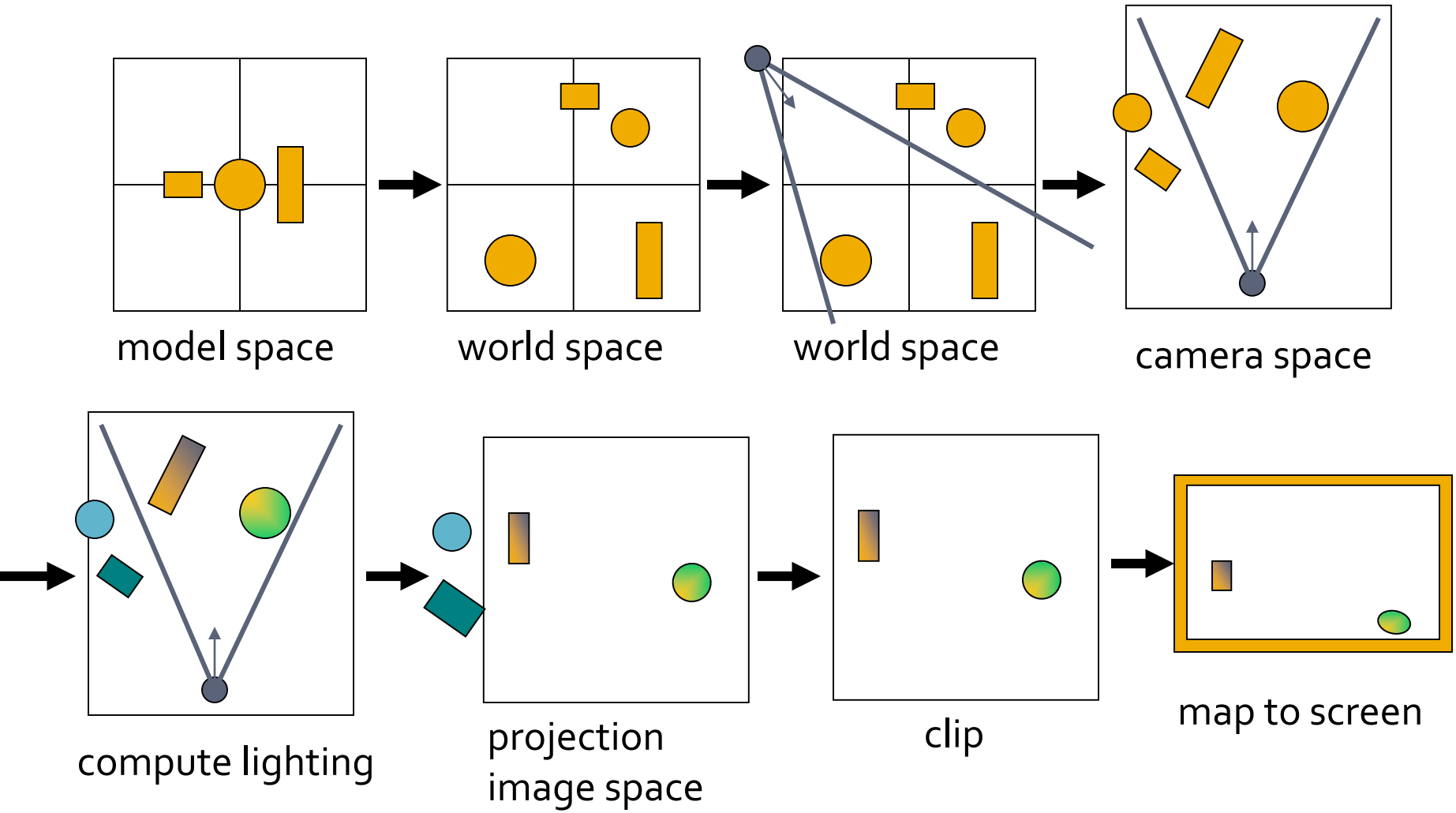
- Matrice di proiezione prospettica 3x4.
- In CG (dx) è usato un modello analogo al pinhole (sx)
- In realtà vedremo come vi sia uno step intermedio: la matrice di projection trasforma il view frustum in un cuboide (matrice 4x4).



Clipping and Screen Mapping

- Dopo la trasformazione di projection, i vertici sono proiettati su un cuboide con x,y in [-1, 1] e z in [0,1].
- I modelli geometrici che cadono fuori tale cuboide vengono rimossi (Clipping).
- Lo Screen Mapping mappa l'output su finestra ("Screen space coordinate").
- "Screen space coordinate" + valori Z spediti al Rasterizer (di tutte le primitive geometriche dentro il View Frustum)

Riassunto Geometry stage

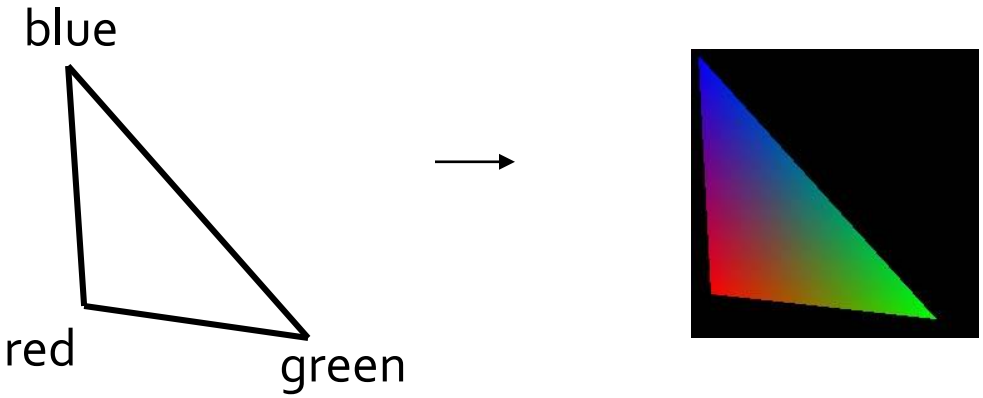


Rasterizer Stage in dettaglio

- Tre Fasi: Triangle Traversal, Pixel Shading, Merging.
- Scan-conversion (Triangle Traversal)
  - Trova i pixel(frammenti) interni di ogni primitiva.
- Interpolazione dei valori dei frammenti interni al triangolo.
- Pixel Shading
  - Calcolo colore, texturing, ….
- Z-buffering, Alpha Blending, Stencil-Buffer, Double buffering (Merging).

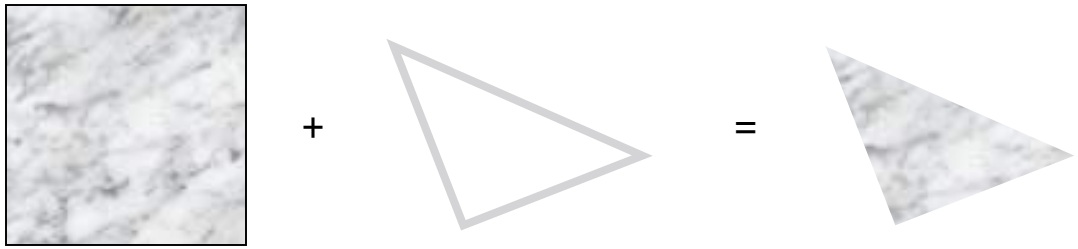
Interpolazione (Rasterizer)

- Interpolazione lineare colori di un triangolo (Gouraud Interpolation)



Texturing

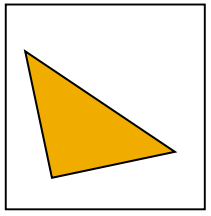
- Incolla le immagini sui triangoli.



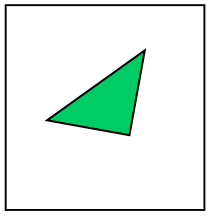
- Alla base di molte molte tecniche avanzate…

Z-Buffering

- La scheda grafica disegna triangoli; di default non gestisce le occlusioni, li disegna nell'ordine in cui arrivano.
- Le occlusioni danno forte indizio di tridimensionalità e realismo (i.e. I triangoli occlusi non devono essere disegnati davanti!)
- Soluzione base: algoritmo del pittore – Ordina oggetti per profondità e spediscili alla scheda video → Non funziona per oggetti intersecanti e ordinare tutti i triangoli richiederebbe troppo tempo.

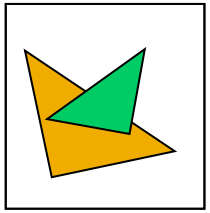


Triangle 1



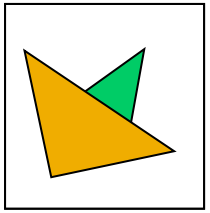
Triangle 2

incorrect



Draw 1 then 2

correct



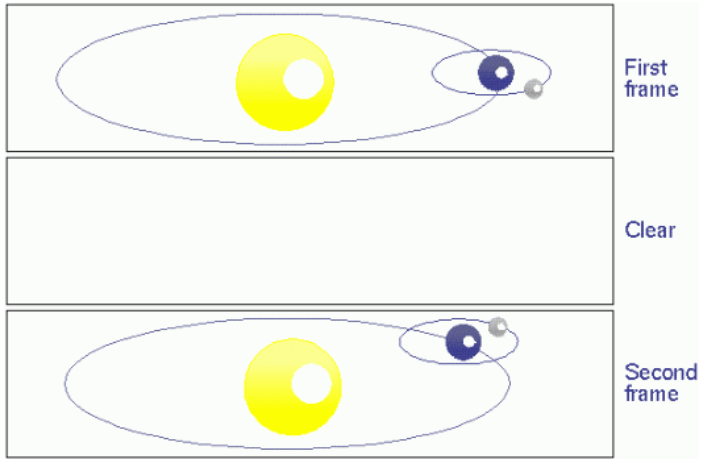
Draw 2 then 1

Z-Buffering

- Soluzione intelligente: Z-Buffering
  - Memorizza la profondità Z di ogni pixel – buffer di dimensioni uguali al frame buffer
  - Durante lo scan conversion viene calcolata una z per ogni frammento di triangolo.
  - Se lo z del frammento è minore dello z del pixel relativo nello Z-Buffer, disegna il frammento nel frame-buffer e aggiorna lo Z-Buffer, altrimenti non scrivere niente.
- Non dobbiamo preoccuparci dell'ordine.
- E' non lineare con la distanza, qualche problema di precisione con oggetti molto distanti.

Double Buffering

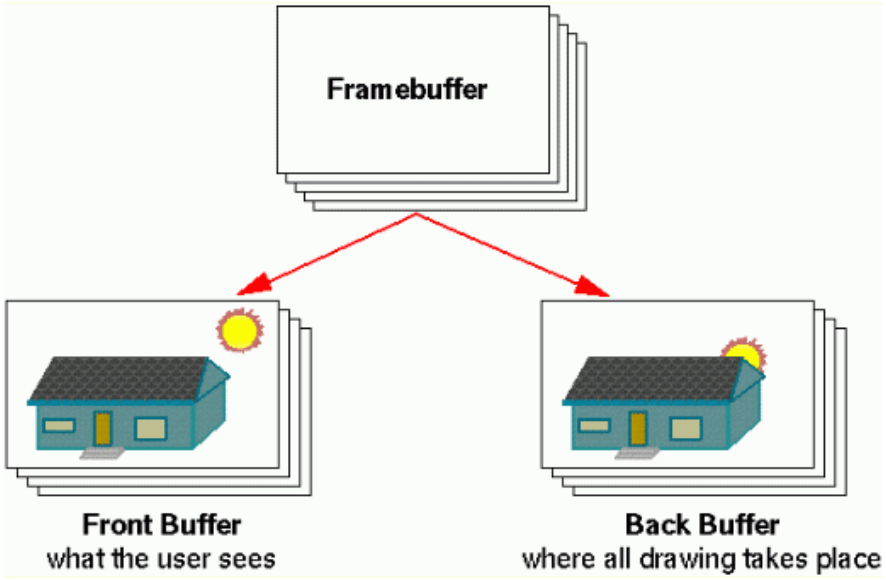
- Il monitor visualizza una sola immagine alla volta. Se facciamo il rendering direttamente sull'immagine, verranno disegnate le primitive appena sono processate.
- Inoltre, l'immagine viene pulita ad ogni frame e quella pulita generata sarà visibile → fenomeno di flickering.



Soluzione : Double buffering

Double Buffering

- Utilizzo di due buffer: uno davanti che viene visualizzato e uno dietro dove vengono effettuate le modifiche.
- Al termine di ogni frame si effettua uno swap.



GPU in dettaglio

- Il termine GPU (Graphics Processing Unit) è stato usato per la prima volta nel 1999 per differenziare la Geforce 256 dalle precedenti schede che eseguivano solo lo stadio di rasterizzazione.
- La GeForce 256 introdusse il Transform-Clipping-Lightning (Geometry stage).

GPU configurabili

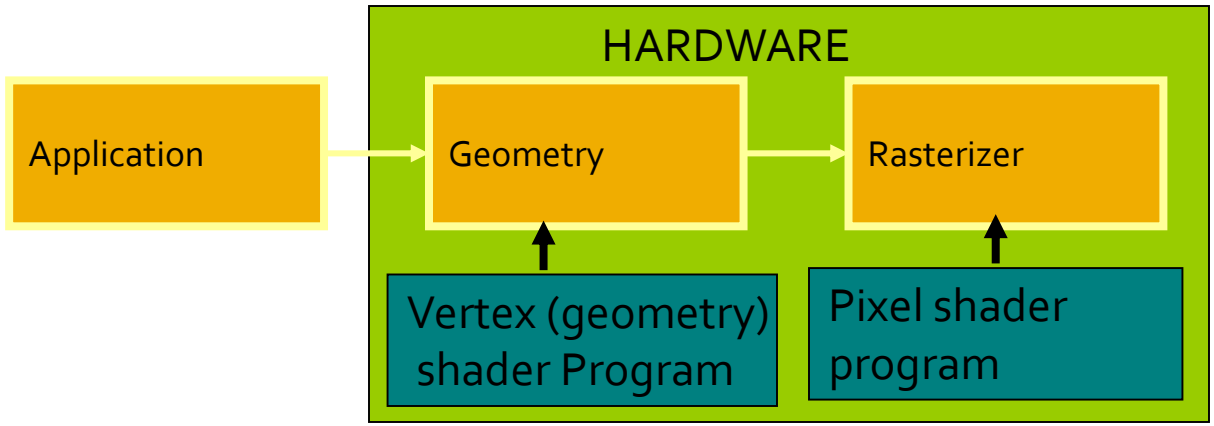
- Nel corso degli anni le gpu si sono evolute. Inizialmente offrivano solo stadi predefiniti che processavano i dati con configurazioni predefinite.
- Successivamente, sempre più operazioni sono divenute altamente configurabili con parametri arbitrari definiti dal programmatore.

GPU programmabili

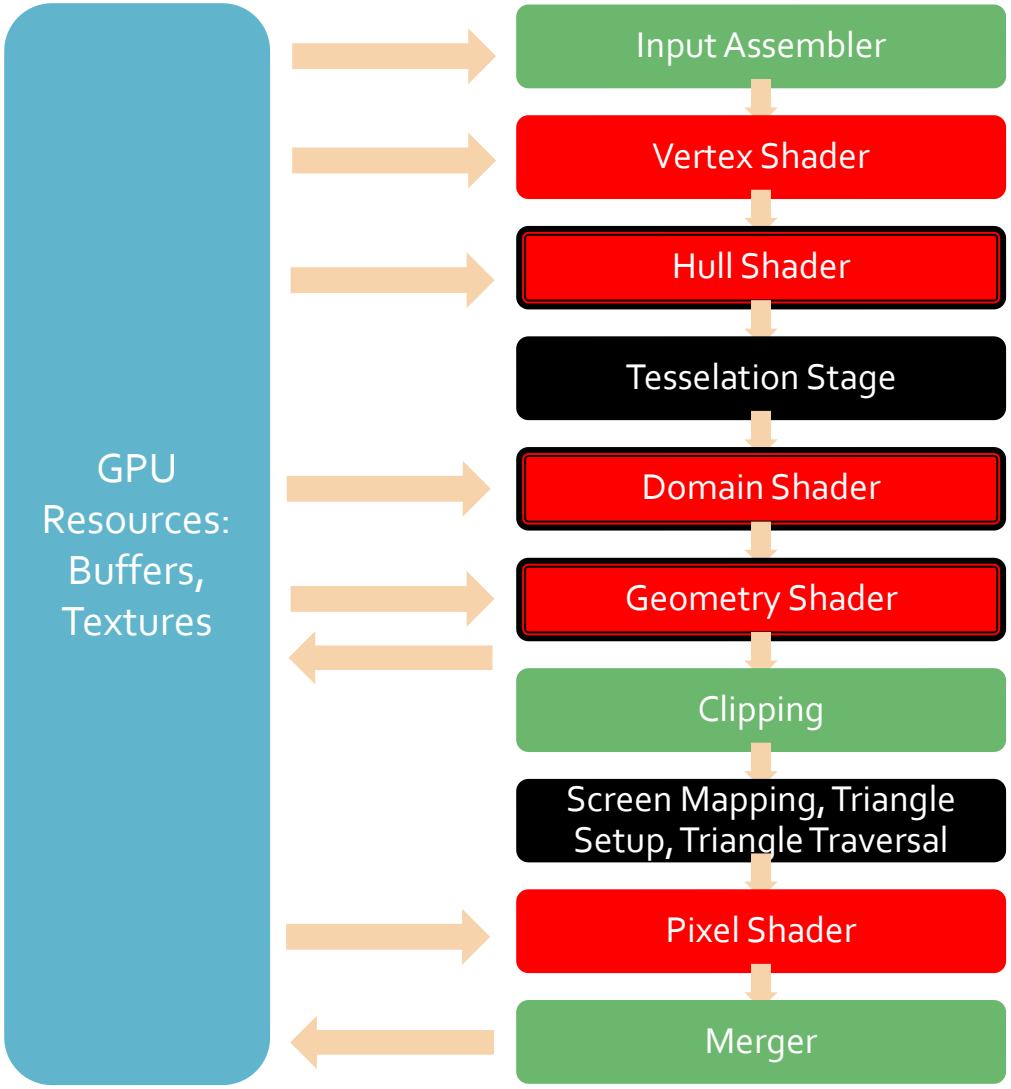
- Dai primi anni 2000 in poi, si sono resi alcuni stadi arbitrariamente programmabili, ovvero è possibile specificare porzioni di programmi da eseguire che rimpiazzano le operazioni predefinite dello stadio (Shaders).
- Il set di istruzioni utilizzabili negli shader è divenuto sempre più esteso nel corso degli anni.

Shaders

- Futuro -> Sempre più libertà al programmatore. Con DX10/11 (OpenGL 2.x) si spinge sempre più la programmazione su shaders rendendola obbligatoria per alcuni stadi.



Stadi principali pipeline GPU attuale

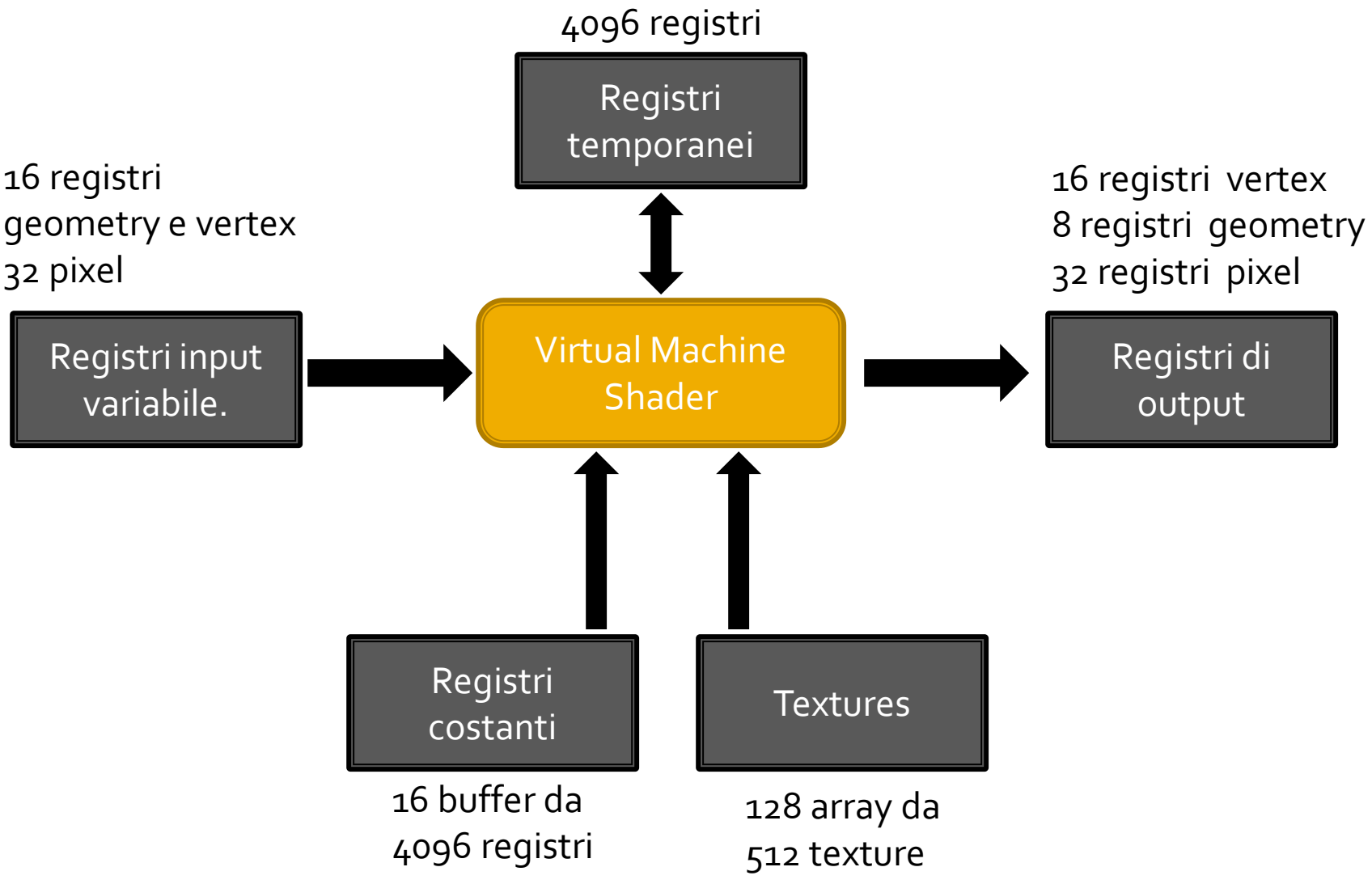


- Gli stadi in rosso sono quelli **programmabili**, quelli in verde **configurabili**, mentre quelli neri sono **fissi**.
- Hull Shader, Tesselation Stage, Domain Shader e Geometry Shader sono facoltativi.

Introduzione shaders programmabili

- Le ultime gpu utilizzano un *Common-Shader Core* (architettura unified).
- Ovvero tutti gli stage programmabili utilizzano hanno un set di istruzioni quasi uguale e sono eseguite nella stessa parte della GPU.
- Vantaggi:
  - Bilanciamento del carico
  - Set di funzioni e limitazioni simili per i vari Shader.

Unified architecture (DX 10/11)



Shader

- Ogni shader ha due tipi di input:
  - *uniform/constant* – input costante per tutte le primitive di una draw call
  - *Varying* – input che varia per ogni vertice o pixel processato
- Inoltre le texture sono come delle grandi matrici bidimensionali read-only.
- Possiamo elaborare dati temporanei nei temporary registers.
- Il tipo di output è prefissato per ogni shader.

Programmazione shaders

- Linguaggio (GLSL, HLSL o Cg) molto simili al C.
  - Supporto sia per controlli di flusso statici (dipendenti da costanti).
  - Sia per flussi dinamici (sono però costosi in termini di performance).
- Le operazioni vettoriali/matriciali floating point sono efficientissime sugli shader.
- Operazioni generiche (sin, cos, radice quadrata, operazioni su interi) sono supportate ma meno efficienti di quelle algebriche.

Confronto modelli shader

Vertex shader version	VS 2.0	VS 2.0a	VS 3.0[2]	VS 4.0[3]
# of instruction slots	256	256	≥ 512	4096
Max # of instructions executed	65536	65536	65536	65536
Instruction predication	No	Yes	Yes	Yes
Temp registers	12	13	32	4096
# constant registers	≥ 256	≥ 256	≥ 256	16x4096
Static Flow Control	Yes	Yes	Yes	Yes
Dynamic Flow Control	No	Yes	Yes	Yes
Dynamic Flow Control Depth	No	24	24	Yes
Vertex Texture Fetch	No	No	Yes	Yes
# of texture samplers	N/A	N/A	4	128
Geometry instancing support	No	No	Yes	Yes
Bitwise Operators	No	No	No	Yes
Native Integers	No	No	No	Yes

Confronto modelli Pixel Shaders

Pixel shader version	2.0	2.0a	2.0b	3.0[2]	4.0[3]
Dependent texture limit	8	Unlimited	8	Unlimited	Unlimited
Texture instruction limit	32	Unlimited	Unlimited	Unlimited	Unlimited
Position register	No	No	No	Yes	Yes
Instruction slots	32 + 64	512	512	≥ 512	≥ 65536
Executed instructions	32 + 64	512	512	65536	Unlimited
Texture indirections	4	No limit	4	No Limit	No Limit
Interpolated registers	2 + 8	2 + 8	2 + 8	10	32
Instruction predication	No	Yes	No	Yes	No
Index input registers	No	No	No	Yes	Yes
Temp registers	12	22	32	32	4096
Constant registers	32	32	32	224	16x4096
Arbitrary swizzling	No	Yes	No	Yes	Yes
Gradient instructions	No	Yes	No	Yes	Yes
Loop count register	No	No	No	Yes	Yes
Face register (2-sided lighting)	No	No	No	Yes	Yes
Dynamic flow control	No	No	No	24	Yes
Bitwise Operators	No	No	No	No	Yes
Native Integers	No	No	No	No	Yes