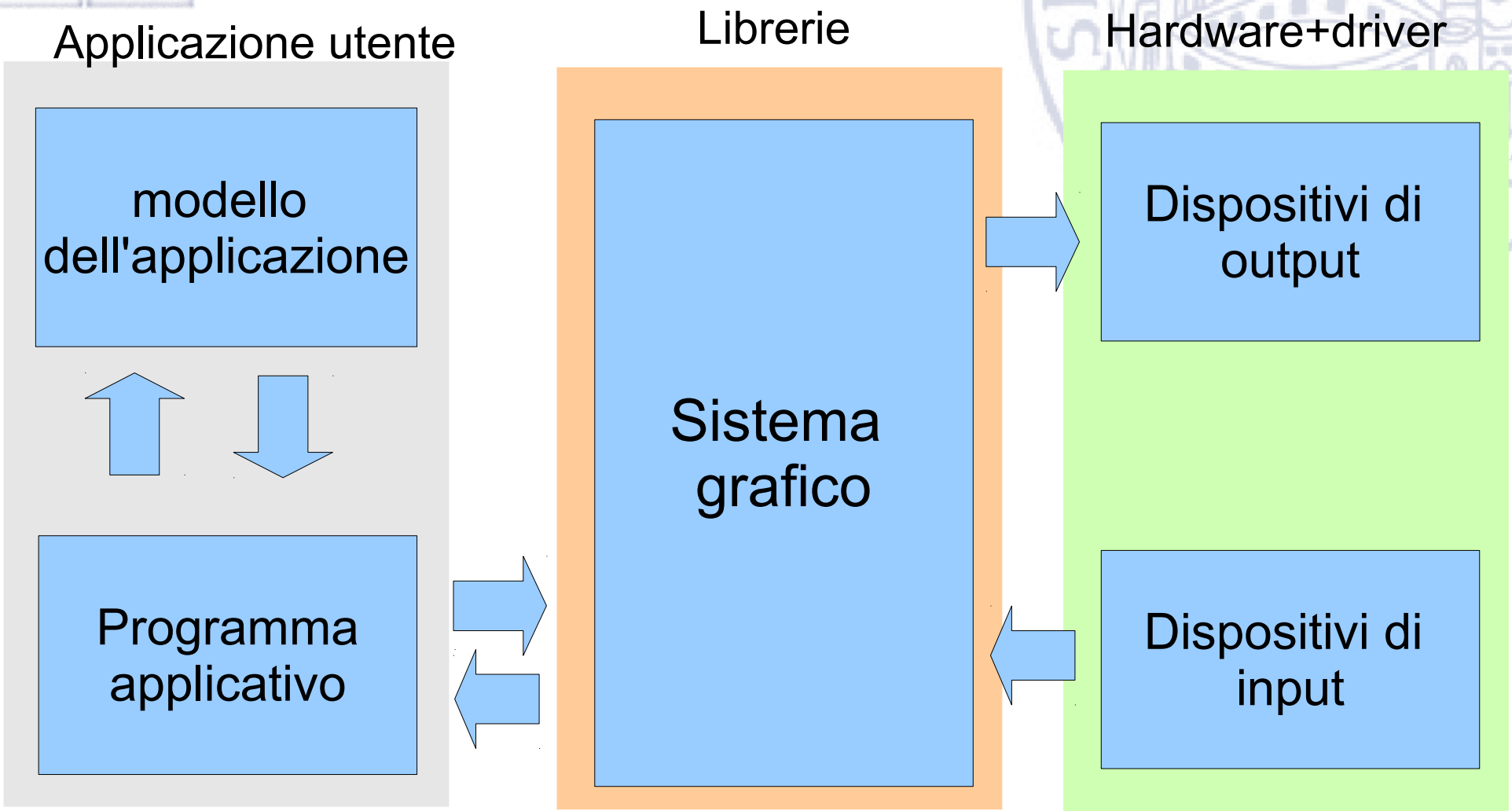
Grafica al calcolatore
OpenGL -1

Andrea Giachetti andrea.giachetti@univr.it
Fabio Marco Caputo fabiomarco.caputo@univr.it
Department of Computer Science, University of Verona, Italy

Avevamo visto: applicazione grafica



Visualizzazione della scena

- Come si implementa la fase di rendering?
  - Ricapitoliamo: creiamo un modello geometrico di scena, descritto da primitive e modelliamo luci e telecamera
  - A ogni istante in cui vogliamo calcolare l'immagine nell'applicazione dobbiamo simulare la formazione dell'immagine e trasferire il risultato all'output
- Applicazioni non interattive:
  - fanno uso di ambienti di rendering più sofisticati e flessibili (ad es. RenderMan), spesso eseguiti SW su cluster di PC
- Applicazioni interattive:
  - si avvalgono pesantemente delle moderne schede grafiche (HW dedicato al processing di dati 3D)

Il sistema grafico

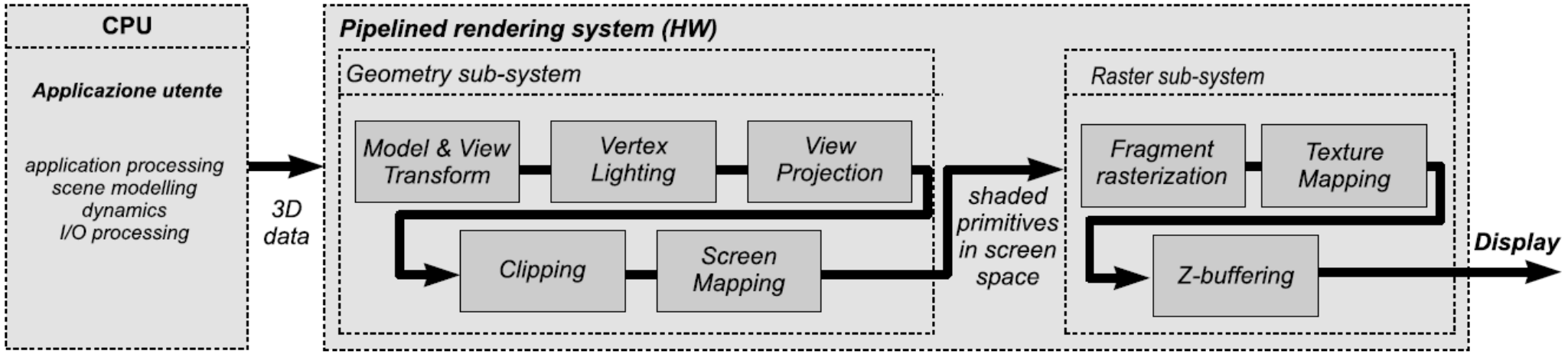
- Le schede grafiche che implementano una procedura di generazione delle immagini a partire da primitive e descrizioni di scena standard, es. OpenGL
- Pipeline di rendering, spesso detta Rasterization: è una semplificazione del processo di formazione immagine fortemente parallelizzata
  - In realtà la rasterizzazione sarebbe solo la seconda parte della pipeline
- Ci arriveremo anche studiando la teoria, capendo quali approssimazioni del modello di illuminazione si fanno e quali trucchi si utilizzano per velocizzare
- Ma cominciamo a utilizzarla attraverso le API in laboratorio

API per la grafica

- API (Application Programming Interface) per la grafica 3D
  - OpenGL, DirectX, …
- Progettate per grafica 3D interattiva, organizzazione logica funzionale ad una efficiente implementabilità HW
- Efficienza direttamente dipendente dalla possibilità di elaborare in parallelo le diverse fasi del processo di rendering
- Soluzione vincente: suddivisione del processo in fasi indipendenti, organizzabili in pipeline
  - Maggiore parallelismo e velocità di rendering
  - Minore memoria (fasi indipendenti -> memoria locale, non necessario conoscere la rappresentazione dell'intera scena ma solo della porzione trattata)

Pipeline di rendering

- Pipeline adottata dalle comuni API grafiche 3D e, conseguentemente, caratterizzante l'architettura dei sottosistemi grafici 3D hardware
- Le singole primitive 3D fluiscono dall'applicazione al sottosistema grafico, e sono trattate in modo indipendente dai vari stadi del sottosistema grafico
- Approccio proposto inizialmente da Silicon Graphics (1982) e poi adottato da tutti i produttori di HW grafico



Pipeline di rendering

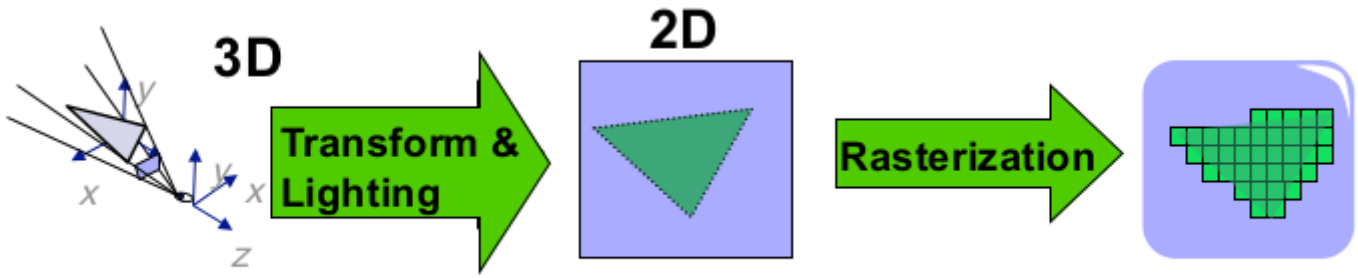
- Tre principali fasi elaborative:
  - gestione e trasmissione della rappresentazione tridimensionale (a cura dell'applicazione)
  - gestione della geometria (Geometry Subsystem)
  - gestione della rasterizzazione (Raster Subsystem)
    - Queste ultime normalmente su GPU

Primitive geometriche

- Le primitive geometriche effettivamente visualizzate in questa pipeline sono poligoni (modelleremo usando questi), di fatto solo triangoli
- L'applicazione ad alto livello avrà la scena definita con strutture dati varie, ma sceglierà ad ogni istante
  - Quali poligoni mandare alla pipeline grafica
  - I parametri relativi alla vista
- Da qui tutto avviene sull'hardware grafico
- La pipeline lavora quindi in object order, cioè si parte dalle primitive e non dai pixel dell'immagine che si vuole ottenere (image order), che, come vedremo in teoria, sarebbe più naturale per creare i pixel delle immagini simulando la fisica dell'interazione luce-materia

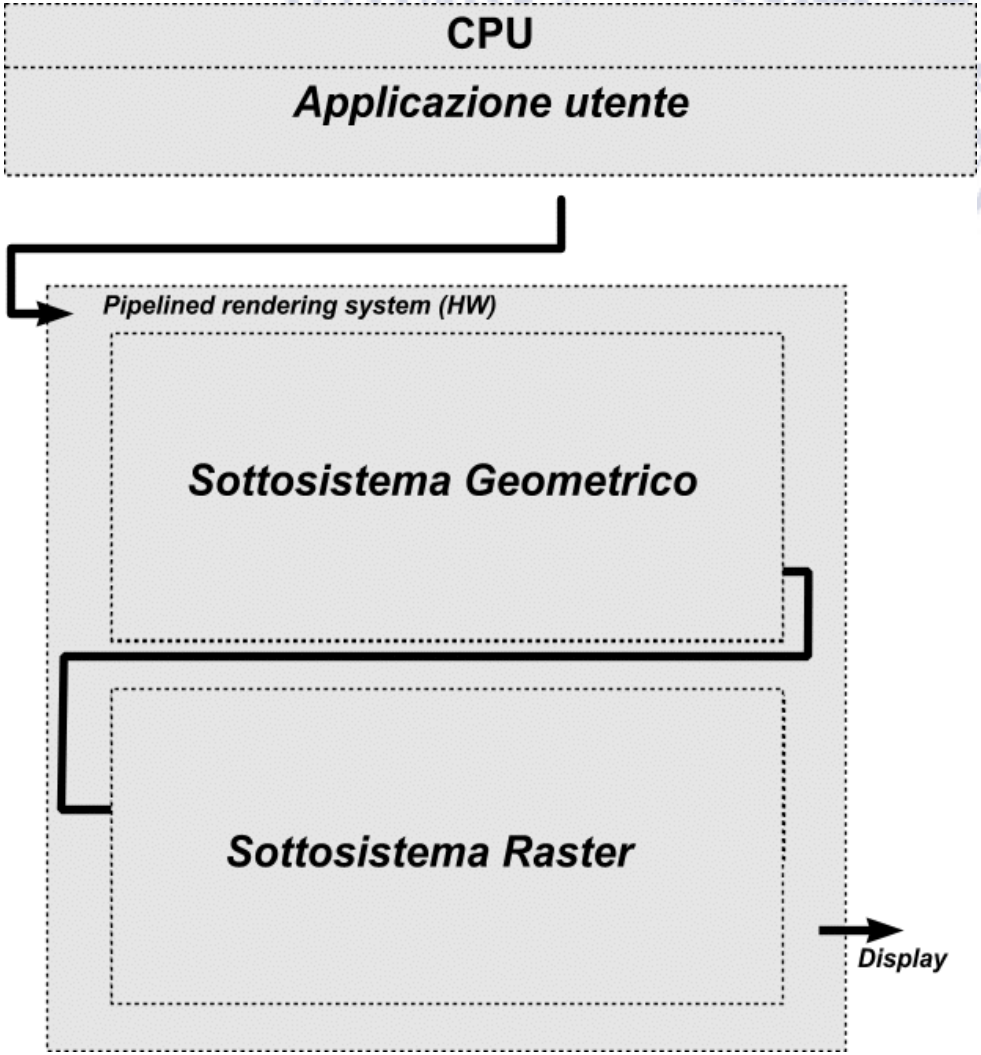
OpenGL Rendering Pipeline

- Nei prossimi lucidi ci concentreremo sul funzionamento della pipeline grafica di rasterizzazione riferendoci in generale alle librerie OpenGL
  - Basata sul concetto di rendering di una scena tramite la proiezione e rasterizzazione in object order di tutte le primitive della scena
  - 2 stadi: geometrico e rasterizzazione



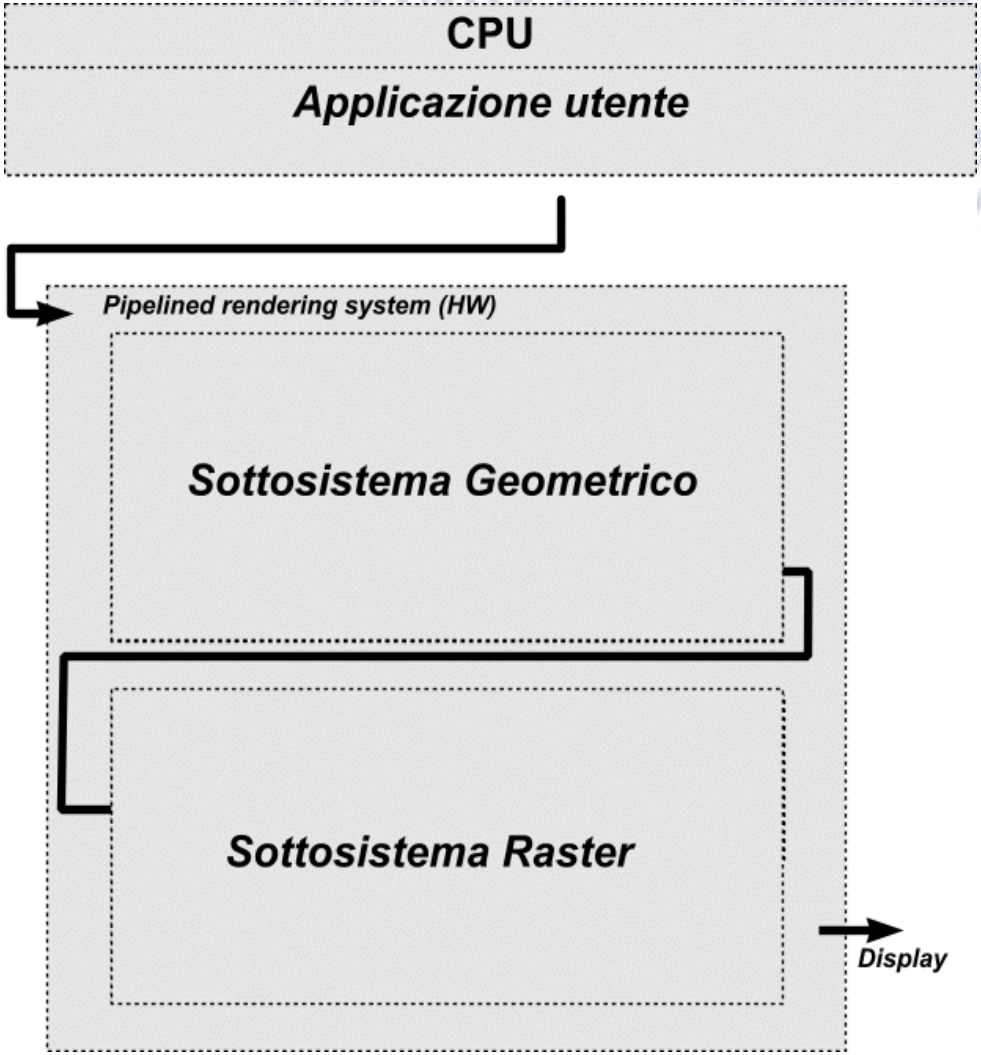
Basic Pipeline

- Tre principali attori in gioco
  - L'applicazione
    - che gestisce la scena e decide quali delle molte primitive che la compongono è necessario mandare agli stadi successivi della pipeline
  - Il sottosistema geometrico
  - Il sottosistema raster



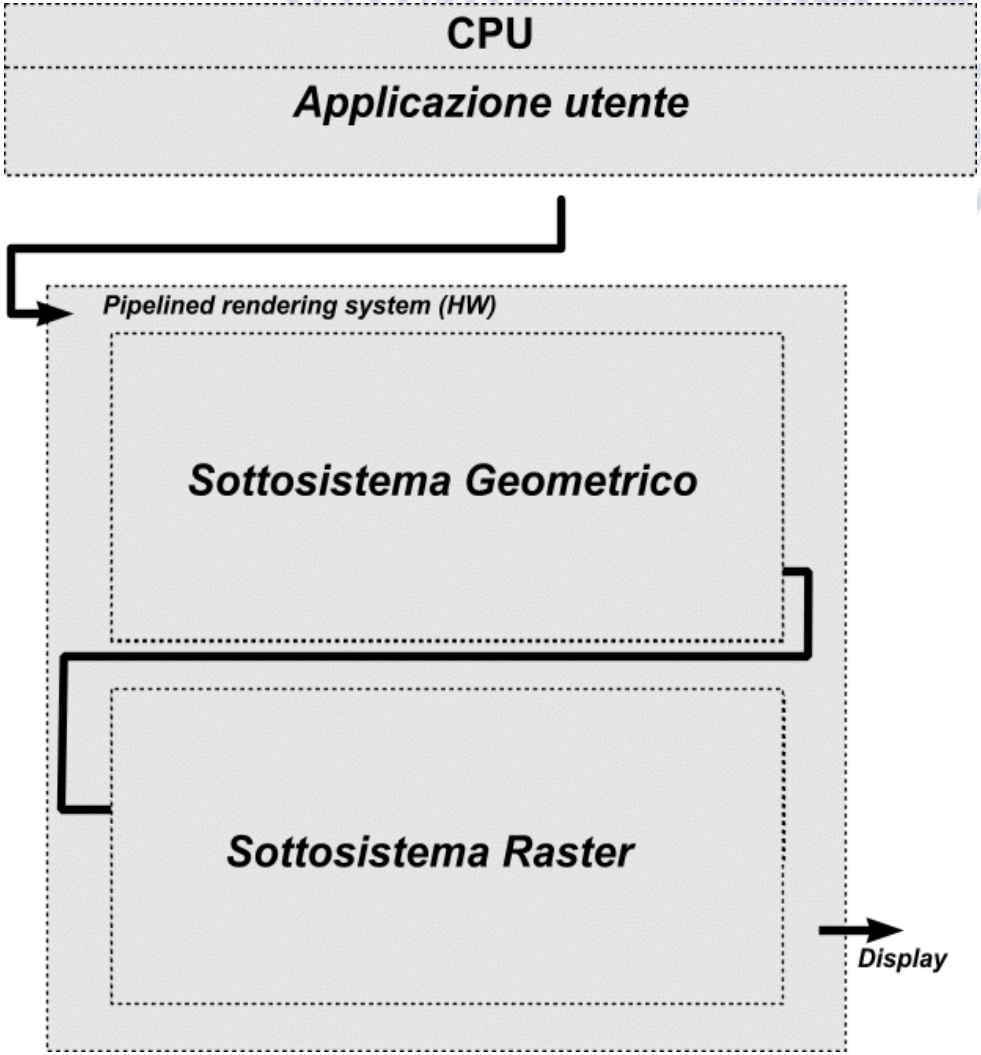
Basic Pipeline

- Tre principali attori in gioco
  - L'applicazione
  - Il sottosistema geometrico
    - che processa le primitive in ingresso e decide
      - se, (culling)
      - come, (lighting)
      - e dove (transf & proj)
    - devono andare a finire sullo schermo
  - Il sottosistema raster



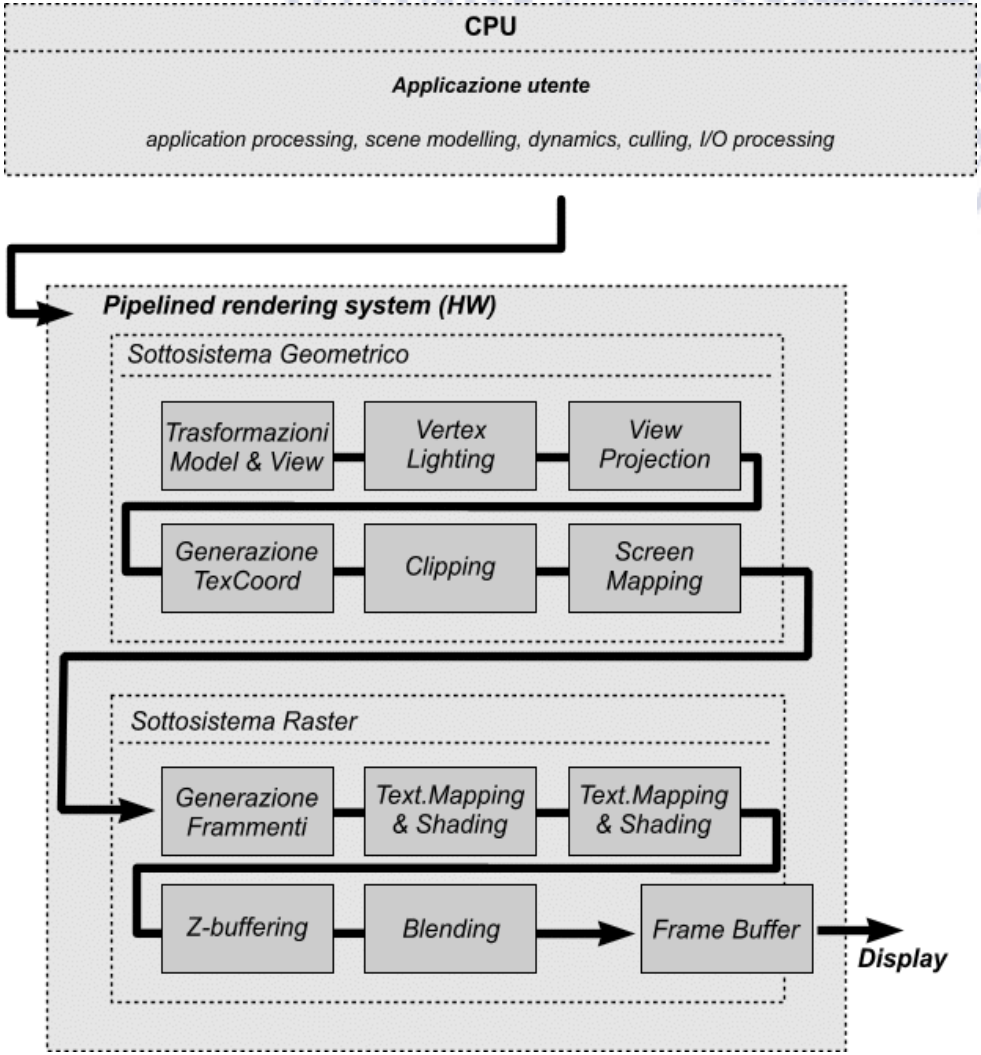
Basic Pipeline

- Tre principali attori in gioco
  - L'applicazione
  - Il sottosistema geometrico
  - Il sottosistema raster
    - che per ogni primitiva di cui ormai si conosce la posizione finale accende i pixel dello schermo da essa coperti in accordo a
      - Colore
      - Texture
      - profondità



Basic Pipeline

- I nostri programmi utilizzeranno la pipeline
- Nelle lezioni di teoria vedremo le informazioni teoriche su
  - Modellazione
  - Fisica della formazione delle immagini

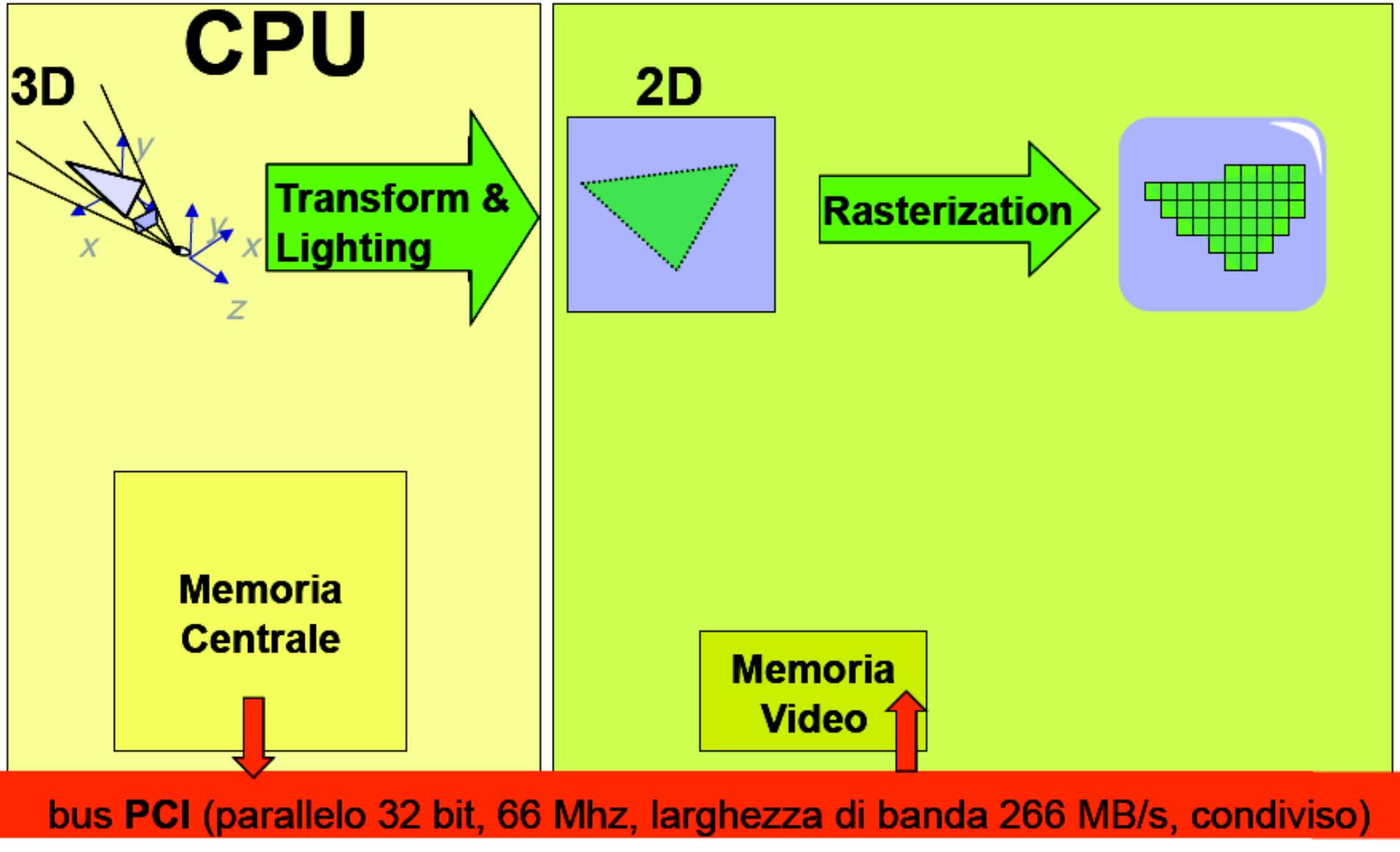


Schede grafiche

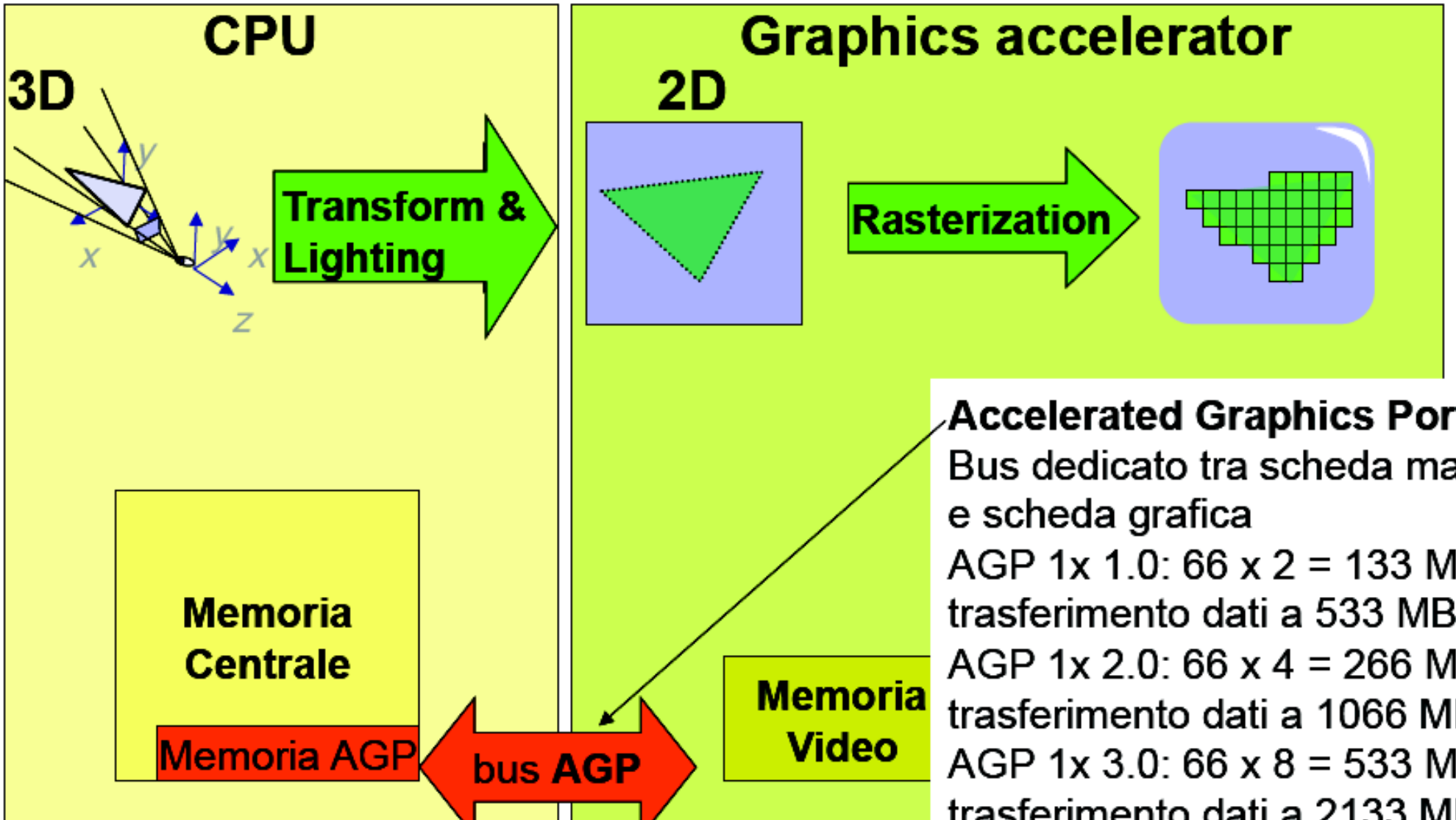
- Gestiscono nei sistemi moderni interattivi tutta la parte di pipeline del sottosistema raster+geometrico
- Non è sempre stato così
- Oggi possono fare molto di più e sono in pratica sistemi di calcolo parallelo
  - Si possono implementare algoritmi di rendering più complessi della pipeline standard
  - Si può fare calcolo generico (GPGPU)

Evoluzione storica

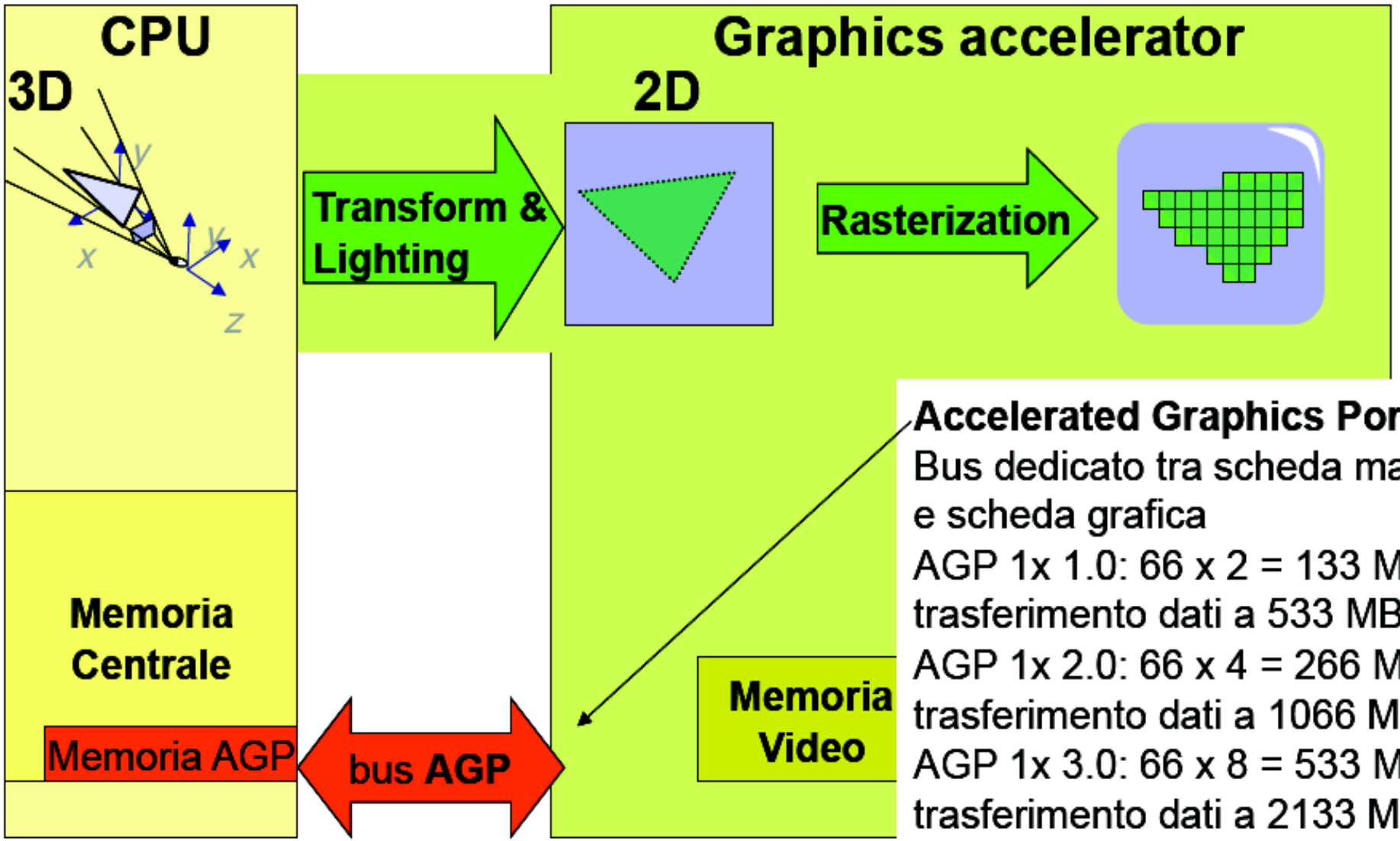
- 1995-1997: 3DFX Voodoo



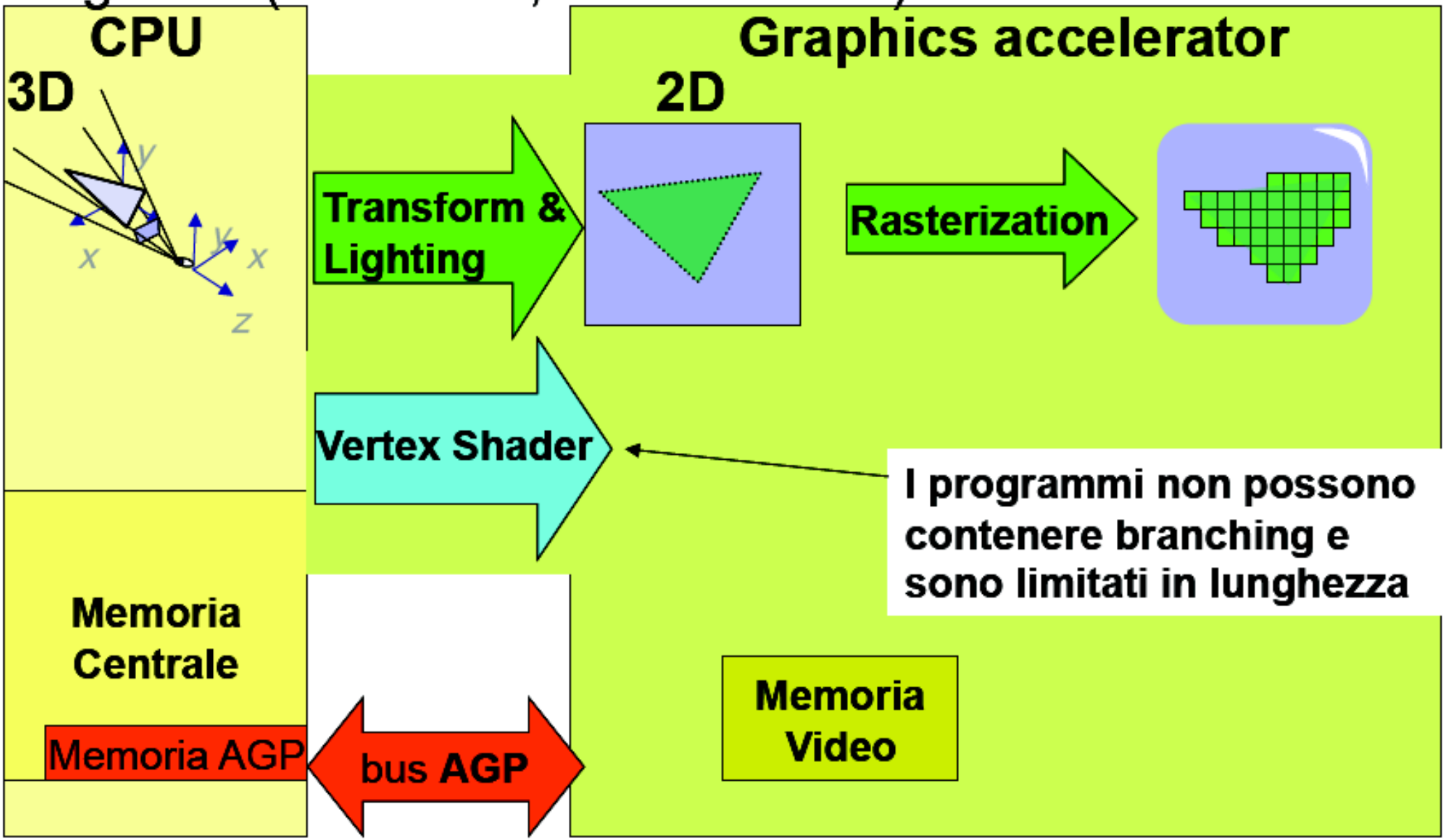
- 1998: NVidia TNT, ATI Rage



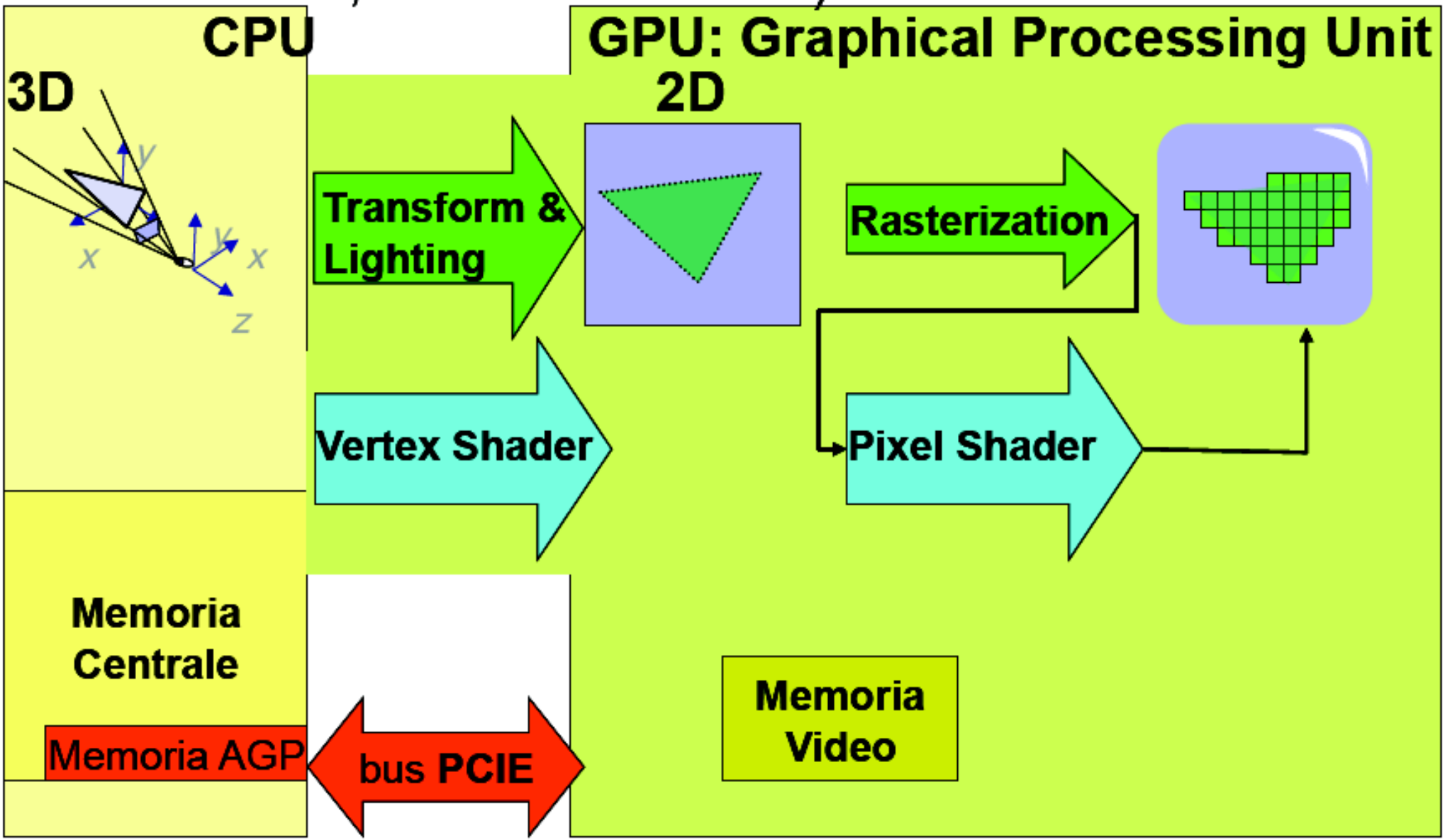
- 1998: NVidia TNT, ATI Rage



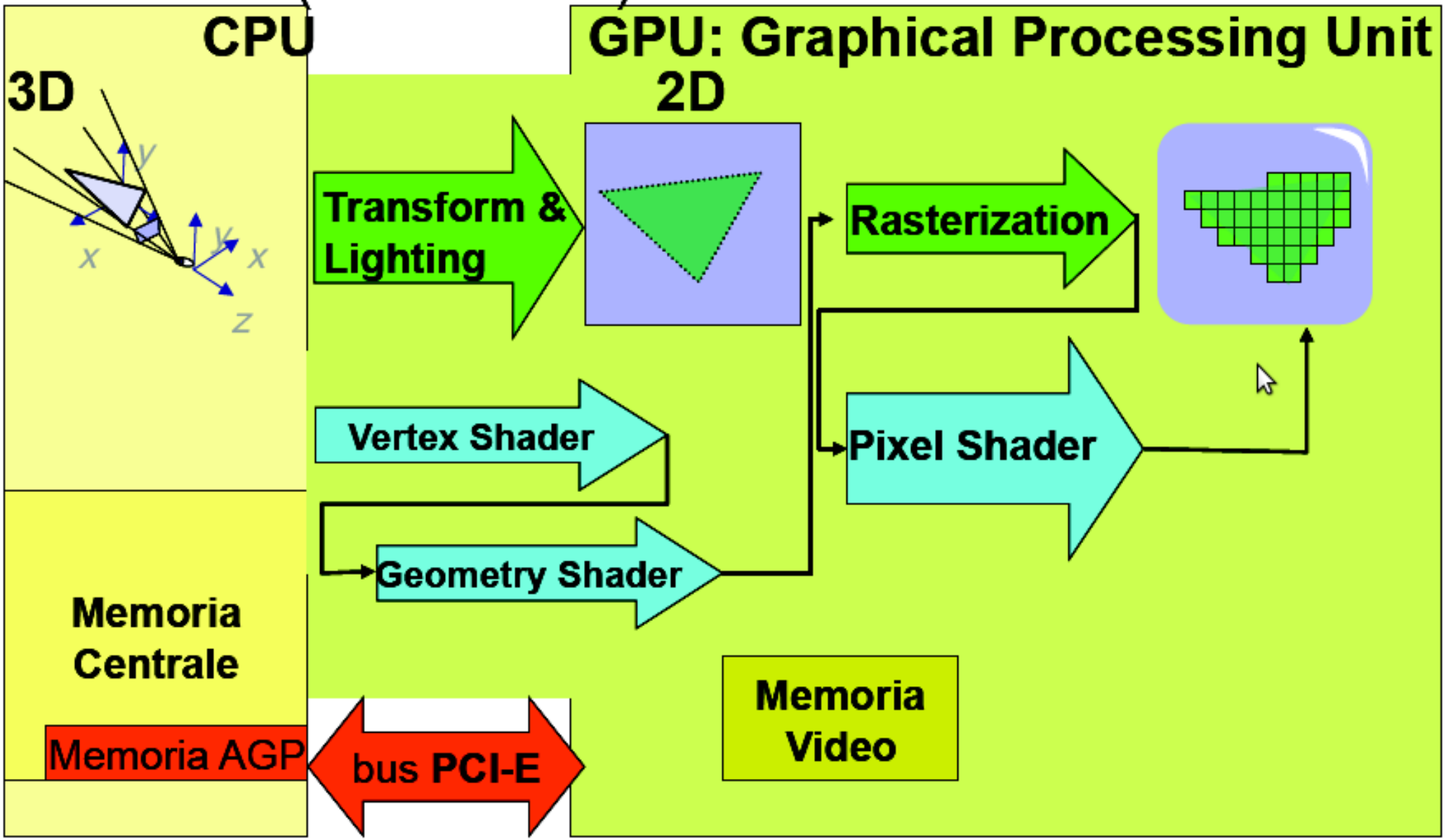
- 2001: Vertex Shader. Programmare il chip della scheda grafica (GeForce3, Radeon 8500)



- 2004-5: PCI-Express, branching negli shader (GeForce 6800-7800, ATI Radeon 9200)



- 2007: geometry shader, stesso hardware per tutti gli shaders (NVidia 8800)



OpenGL

OpenGL = Open Graphics Library

- API (anzi semplicemente una specifica) aperta Standard industriale per il disegno su hardware accelerato
- Implementato dai produttori di hardware grafico
- Versione corrente 4.4 (luglio 2013)
- Bindings per innumerevoli linguaggi di programmazione: C, C++, C#, Java, Fortran, Perl, Python, Delphi, etc.

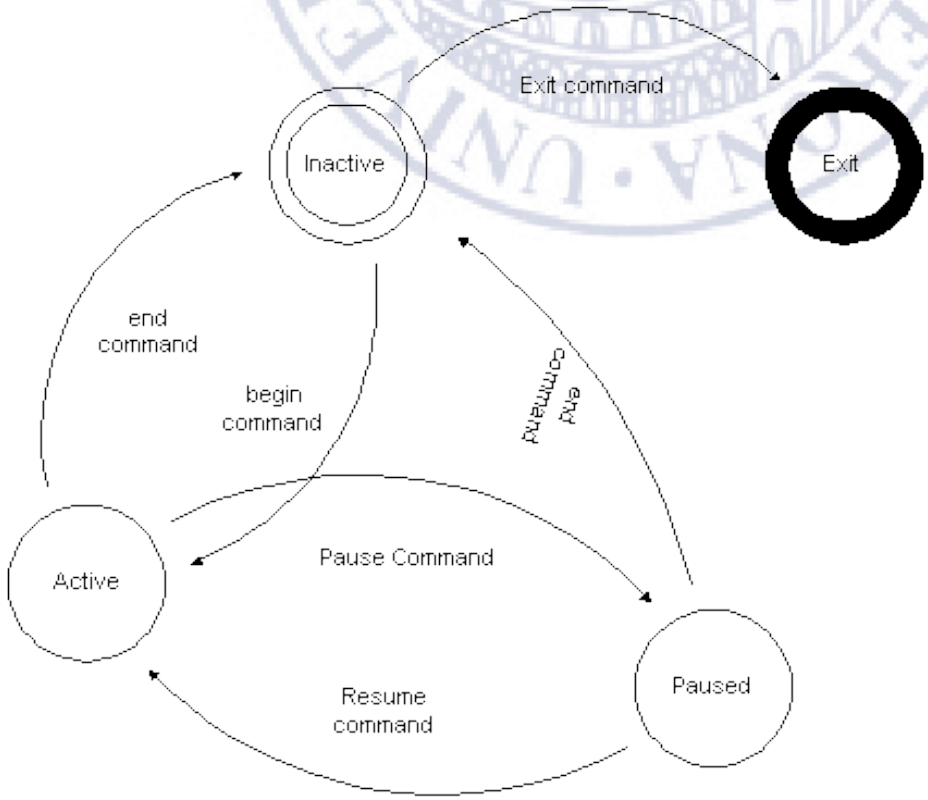
OpenGL

- Creato da Silicon Graphics
- Oggi OpenGL è scritta dall' Architectural Review Board (ARB) parte del Khronos Group

KHRONOS GROUP

Filosofia

- OpenGL è una state-machine
- Quando uno stato è raggiunto, rimane attivi fino a una nuova transizione
- Una transizione in OpenGL è una chiamata a funzione
- Uno stato è definito dagli oggetti OpenGL correnti



Esempio

Set OpenGL-states:

```
glEnable(...);
glDisable(...);
gl*(...);  // several call depending on purpose
```

Query OpenGL-states with Get-Methods:

```
glGet*(...);  // several calls available, depending
on what to query
```

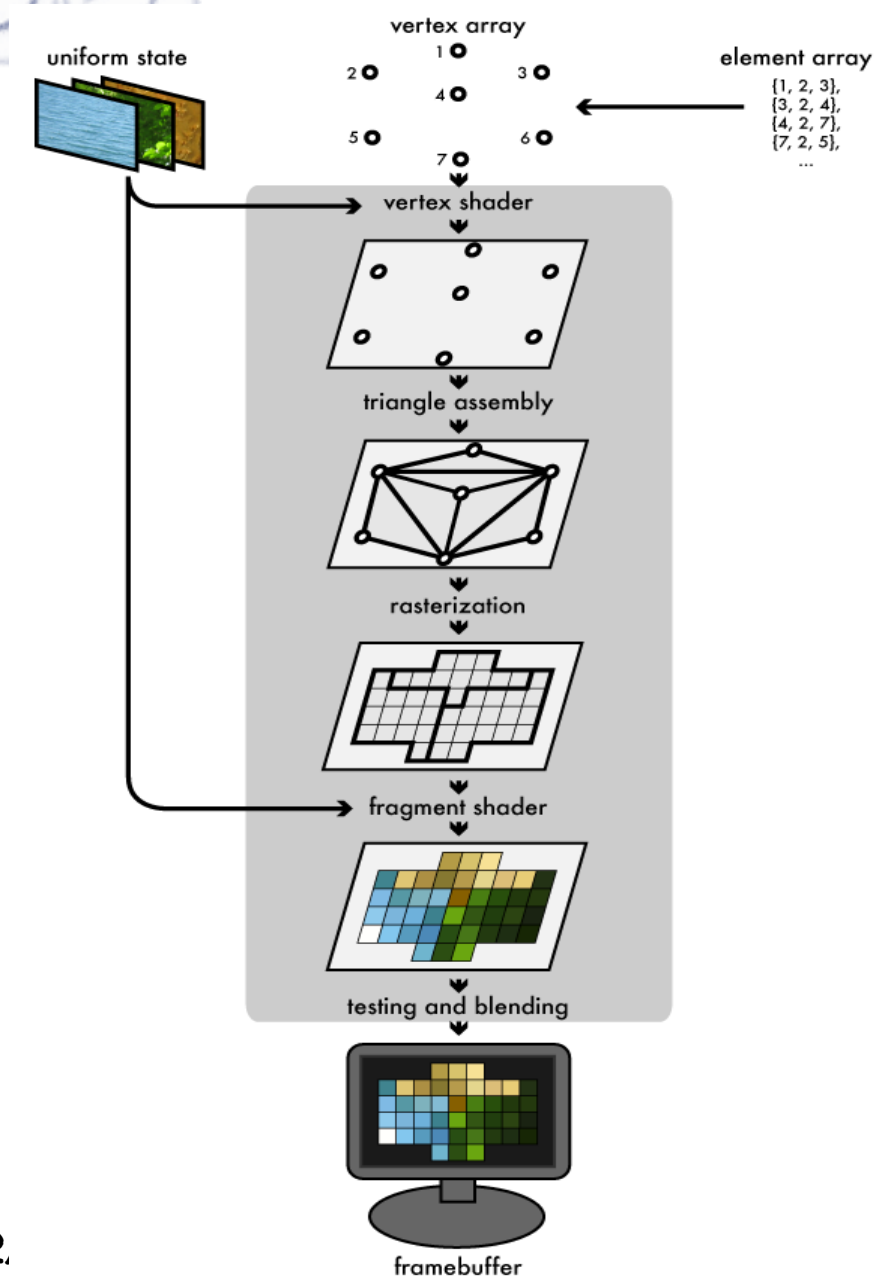
Evoluzione

- In OpenGL 1.0-1.4 si crea una pipeline di base che descriveremo durante il laboratorio, con disegno, shading, texture, ecc.
  - Fixed function pipeline: ci sono motori separati per trasformazione geometrica e illuminazione per supportare l'accelerazione hardware del calcolo della geometria e dell'illuminazione
  - Le funzioni della GPU non sono programmabili e si possono solo configurare cambiando i parametri delle funzioni OpenGL

Evoluzione

- Le GPU moderne integrano processori unificati che possono processare pù stadi e possono essere programmate con linguaggi specifici come GLSL
- OpenGL 2.1 (2006) mescola fixed function e programmabilità

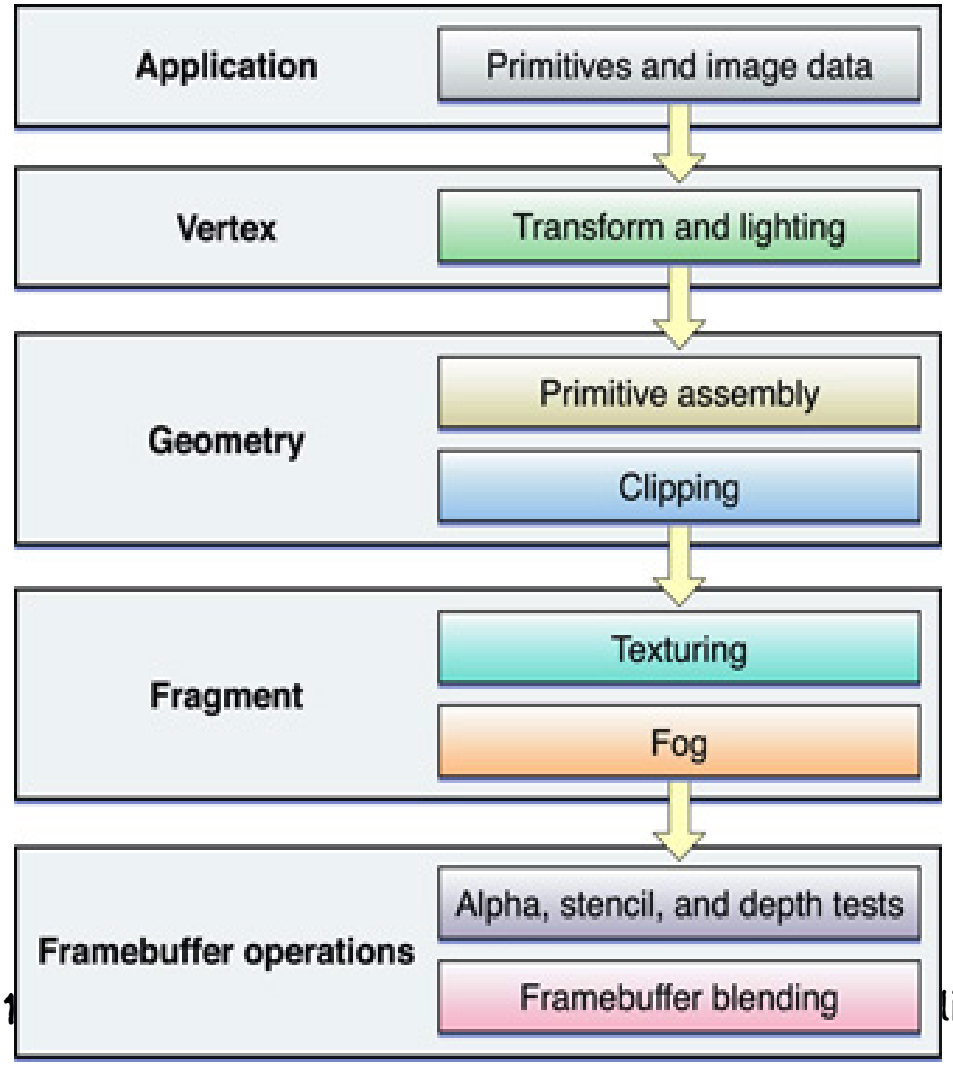
La pipeline grafica "standard"



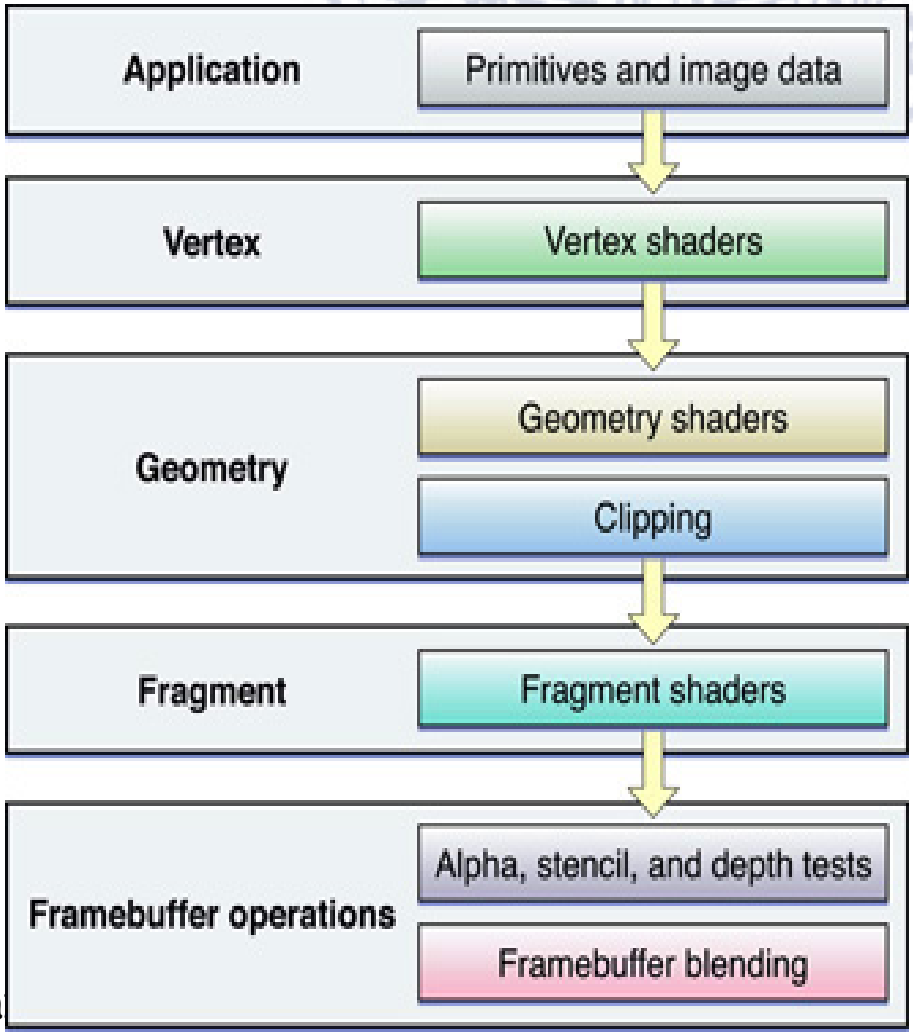
- Prima processing geometrico sui vertici
- Poi rasterizzazione e composizione dell'immagine bitmap
- Sono state implementate funzioni fisse standard da accelerare in due pipeline sequenziali
- Poi tutto è mutato, ma la pipeline "fissa" resta usabile

Pipeline

- Fissa



- Con shaders

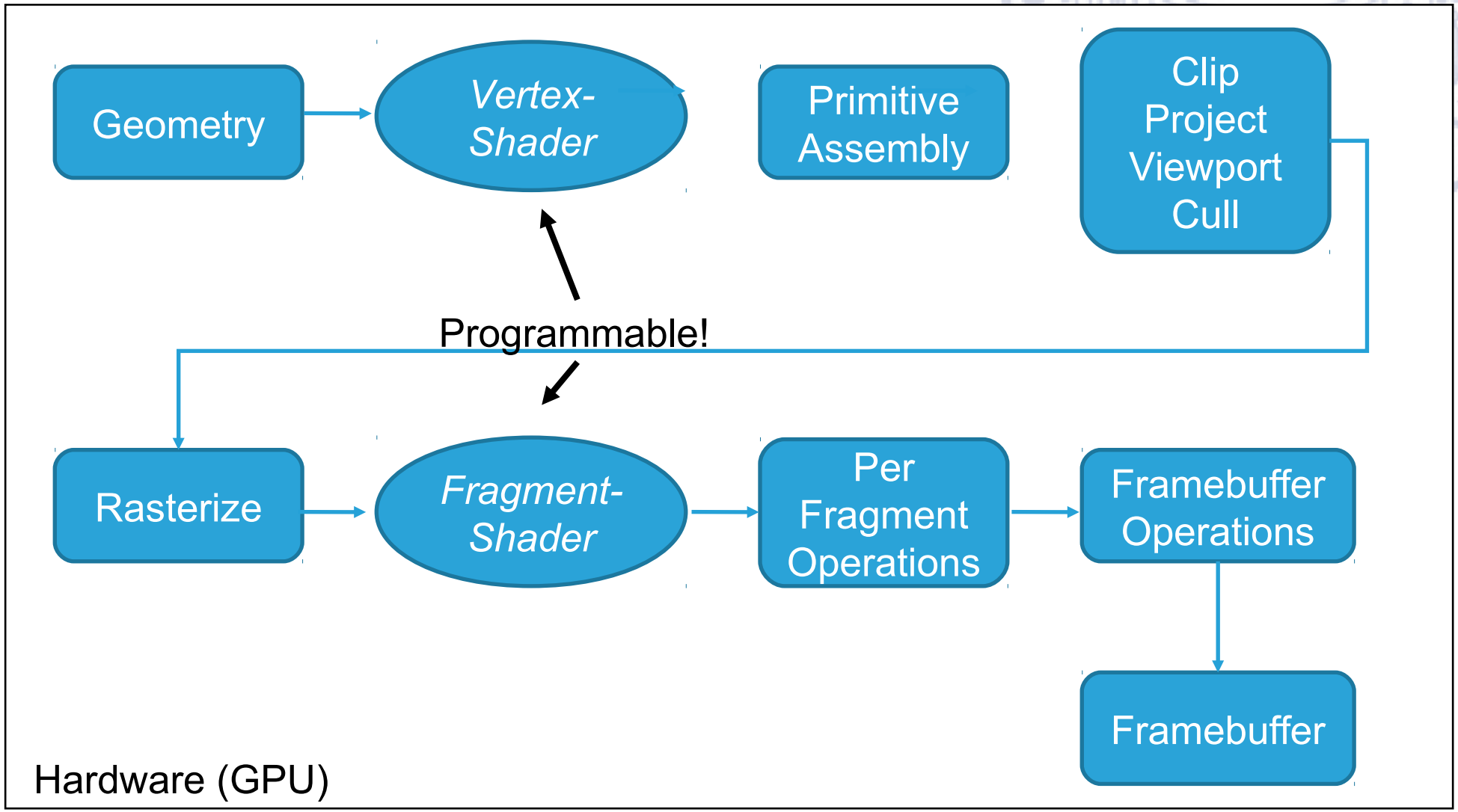


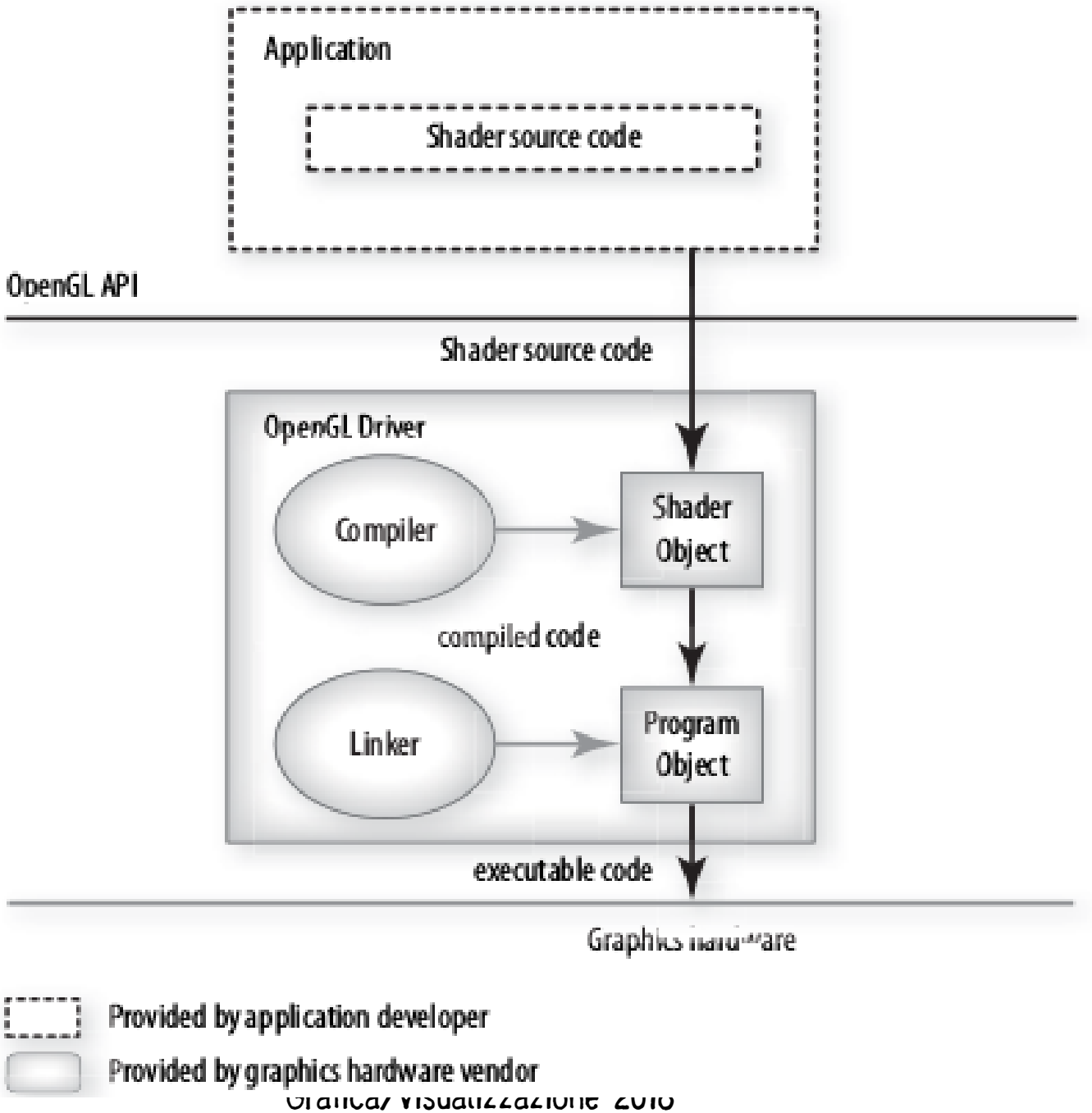
OpenGL

- Tutto è evoluto nelle varie versioni delle OpenGL Specification. A differenza di Direct3D il codice però continua a funzionare nelle versioni successive di OpenGL.
- OpenGL 3.0 (2008) ha reso “deprecated” molte vecchie funzioni e 3.1 ne ha in teoria rimosse molte dividendo la specifica in due: core and compatibility.
  - Core con solo le nuove funzioni compatibility con la vecchia
  - In pratica nessun driver OpenGL implementa soltanto il profilo “core”

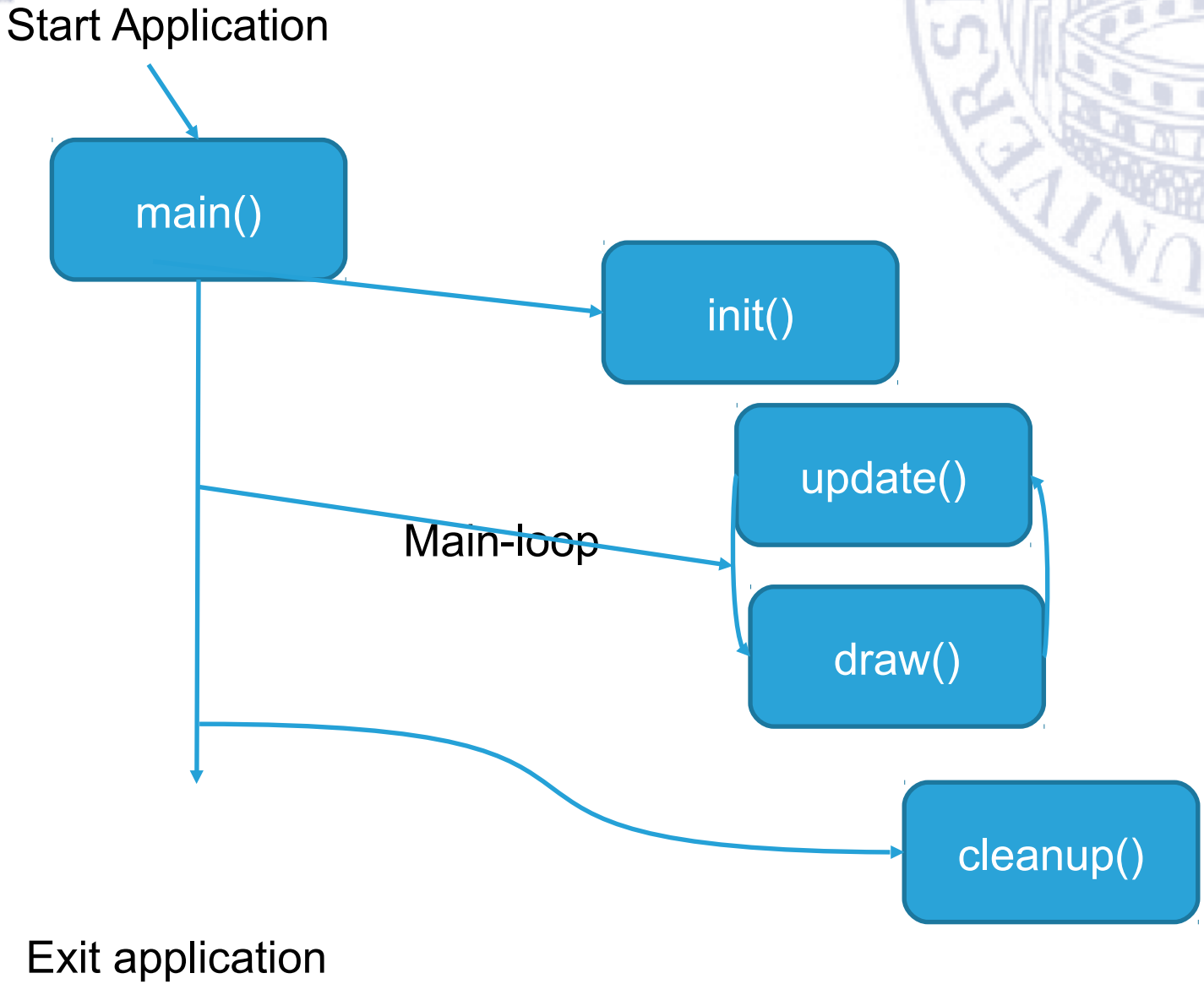
2.1	3.0	3.1	3.2/3.3/4.0
FF	Deprecated Features and Non-FWD-CC	"GL_ARB_ compatibility" extension	Compatibility-Profile

OpenGL 3.x Rendering-Pipeline:




Application
Shader source code
OpenGL API
Shader source code
OpenGL Driver
Compiler
Shader Object
compiled code
Linker
Program Object
executable code
Graphics hardware
Provided by application developer
Provided by graphics hardware vendor

Tipica applicazione openGL

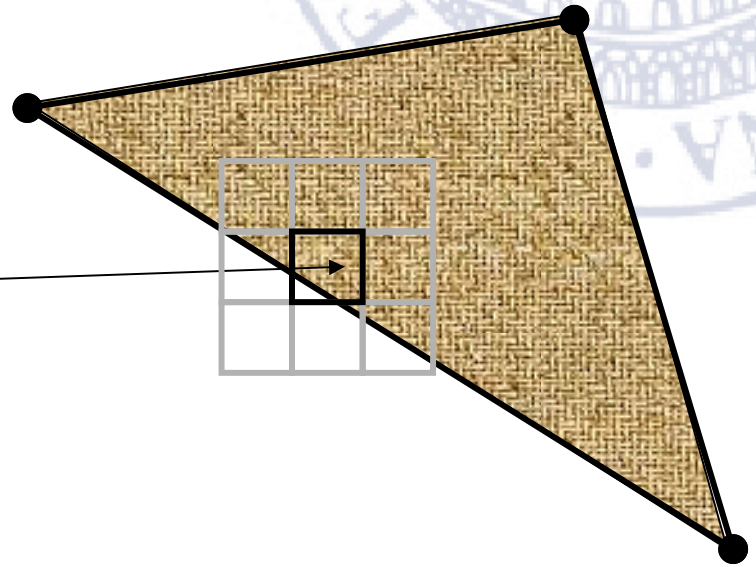


Shaders

- Piccoli programmi scritti in linguaggio simile al C (GLSL), eseguiti sull'hardware grafico
- Sostituiscono la pipeline a funzioni fisse
  - Più difficile da imparare all'inizio, ma potente
- Tipi
  - Vertex Shader (VS): per vertex operations
  - Tessellation Shader (TS) da 4.3
  - Geometry Shader (GS): per primitive operations (da 3.2)
  - Fragment shader (FS): per fragment operations
- Qui si implementano le trasformazioni, l'illuminazione, ecc.

Shaders

- Gli shaders permettono di scrivere le operazioni svolte per *vertex* e *fragment* (pixel o pixel parziale derivante da interpolazione da vertici



- Gli shaders permettono di scrivere le operazioni svolte per *vertex* e *fragment* (pixel o pixel parziale derivante da interpolazione da vertici

Cosa è programmabile

Per vertex:

- Trasformazione geometrica
- Trasformazione proiettiva
- Normali
- Coordinate texture e loro trasformazione
- Illuminazione
- Applicazione di colore per materiale

Per fragment (pixel):

- Elaborazione dei valori interpolati sulla griglia (frammenti)
- Accesso alle texture
- Applicazione di texture
- Effetto nebbia
- Somma di colori
- O anche:
  - Pixel zoom
  - Scalatura e traslazione
  - Lettura di tabelle di lookup per il colore
  - Filtraggi (convoluzione)

Shader

- Il Vertex-Shader è eseguito una volta sola per ciascun vertice
- Il Fragment-Shader è eseguito una volta sola per ogni frammento rasterizzato (~ pixel)!
- Sono da pensare come paralleli! E' importante: le GPU sono massivamente parallele
- Esistono diversi linguaggi di programmazione degli shader
  - *HLSL*, the *High Level Shading Language*
    - Author: Microsoft
    - DirectX 8+
  - *Cg*
    - Author: nvidia
  - *GLSL*, the *OpenGL Shading Language*
    - Author: the Khronos Group, a self-sponsored group of industry affiliates (ATI, 3DLabs, etc) (useremo questo)

Shader

- Gli shaders “conoscono” lo stato di rendering (GLSL conosce lo stato dell’OpenGL) e possono accedere alle textures
- Una passata di rendering è intesa come l’elaborazione di una scena da parte del vertex shader e del pixel shader
  - Possono essercene diverse (rendering multi-pass) una passata può inviare il risultato del rendering su una texture per essere utilizzato dalle successive passate (render-to-texture)
- Avviene una compilazione a run-time nel linguaggio assembly della scheda grafica
- Il compilatore prende come argomenti degli array di stringhe, è quindi possibile la manipolazione per il riutilizzo di frammenti di codice
- Nel primo esempio definiremo il programma de passare al compilatore come array di stringhe all'interno del programma

Shader

- Una volta creati degli shader e compilati è possibile creare un programma di shading
- La funzione per creare un programma di shading è:
  glCreateProgram
- Ad un programma di shading si possono attaccare (attach) gli shaders definiti negli shader objects glAttachShader(..).
- Si possono avere più shader attaccati allo stesso programma così come si possono avere più sorgenti in un programma C.
- Vedremo e commenteremo tutto nei codici di esempio

Pulizia

- I programmi si devono prendere cura della pulizia della memoria
  - Comunque potete vedere come si fa negli esempi
- Gli oggetti con cui sono stati creati gli shader sono distrutti subito (c'è il riferimento al compilato)
  - glDeleteShader(vertexShader);
  - glDeleteShader(fragmentShader);
- Per la rimozione del codice compilato degli shader
  - glDeleteProgram(shaderProgram);

Shader

- Devono avere un metodo `main()`
- Il Vertex-Shader deve dare in output almeno `gl_Position`
- Le variabili possono essere definite dall'utente

```
//preprocessor directives like:
#version 150

variable declarations

void main()
{
      do something and write into output variables
}
```

Shader

- Esempi di variabili:

```
uniform  mat4 projMatrix;  // uniform input
in       vec4 vertex;      // attribute-input
out      vec3 fragColor;   // shader output
```

- Tre tipi:
  - *uniform*: non cambiano sulle primitive; read-only
  - *in*: VS: diverse per vertice, read-only;
    FS: interpolate; read-only
  - *out*: shader-output; VS per FS; FS output.

Variabili uniform

- Variabli uniform:

```
// first get location
projMtxLoc = glGetUniformLocation(programHandle,
"projMatrix");

// then set current value
glUniformMatrix4fv(projMtxLoc, 1, GL_FALSE,
currentProjectionMatrix);
```

  - Una variabile di tipo uniform è costante rispetto ad una primitiva ossia non può modificare il proprio valore tra una chiamata a glBegin ed una a glEnd. Le variabili di tipo uniform possono essere lette ma non scritte dal vertex o fragment shaders. La funzione per recuperare la location all'interno di un programma di shading di una variabile uniform (dato il suo nome) è: `glGetUniformLocation`
  - Poi si scrive con `glUniformMatrix4fv` (o con l'equivalente per tipo diverso)

Tipi di dati

- Floating point, interi e boolean sono disponibili:
  - Float
  - Bool
  - int
- Vettori a 2,3 o 4 componenti:
  - vec2 , vec3, vec4 (vettori di float)
  - bvec2, bvec3, bvec4 (vettori di boolean)
  - ivec2, ivec3, ivec4 (vettori di interi)
- Matrice quadrate 2x2, 3x3 e 4x4:
  - Mat2
  - Mat3
  - mat4

Nota

- In GLSL non c'è il casting automatico dei tipi, ma ogni variabile deve essere utilizzata seguendo le intestazioni formali delle funzioni
- I passaggi da un tipo all'altro devono essere esplicite
  - float f = 2.3;
  - bool b = bool(f);
  - float f = float(3); // integer 3 to floating-point 3.0
  - float g = float(b); // Boolean b to floating point
  - vec4 v = vec4(2);
    - // set all components of v to 2.0

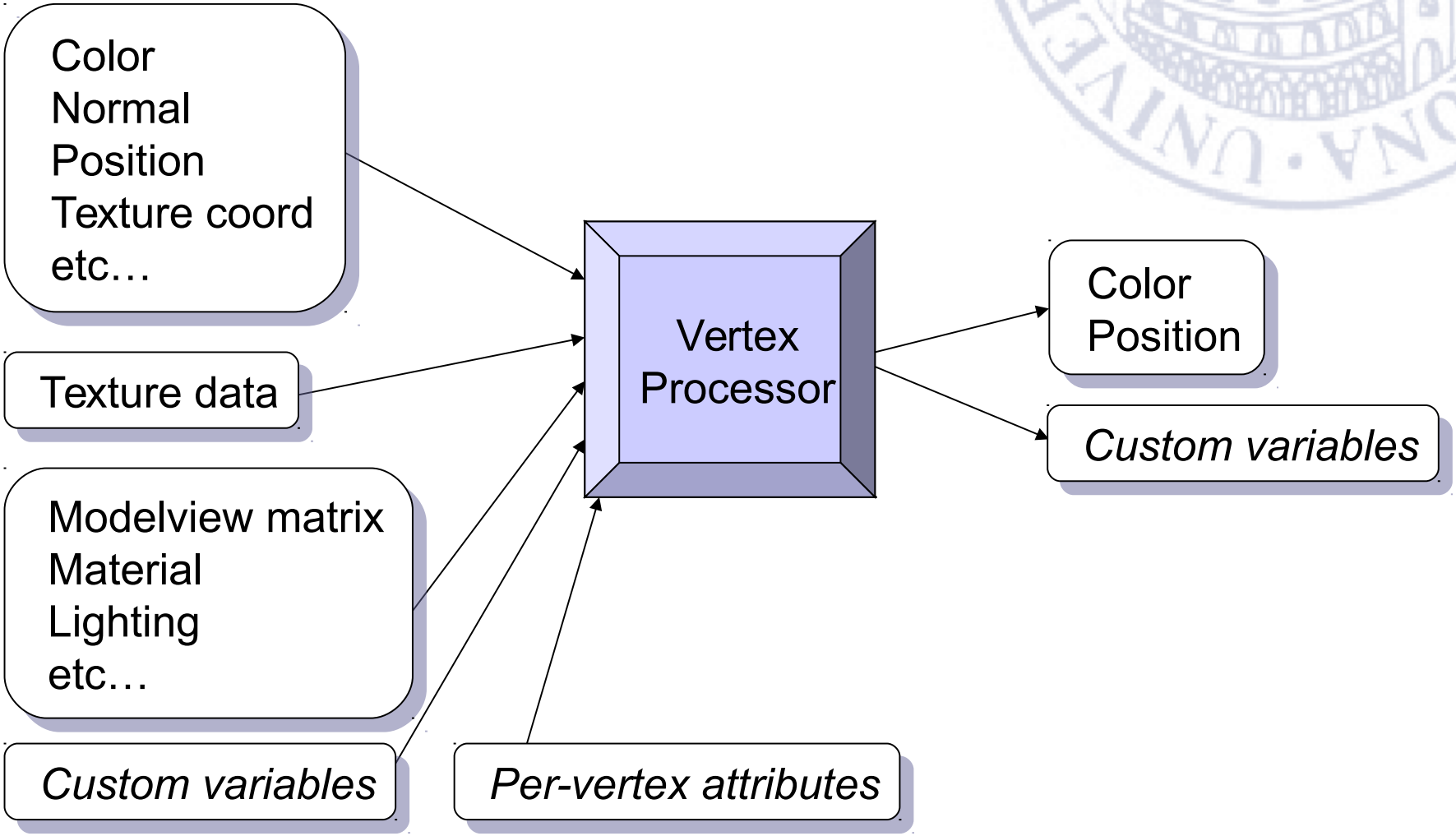
Come trasferisco i vertici?

- I vertici possono avere attributi come ad esempio normali o coordinate di texture
- OpenGL tratta anche le coordinate stesse come attributi

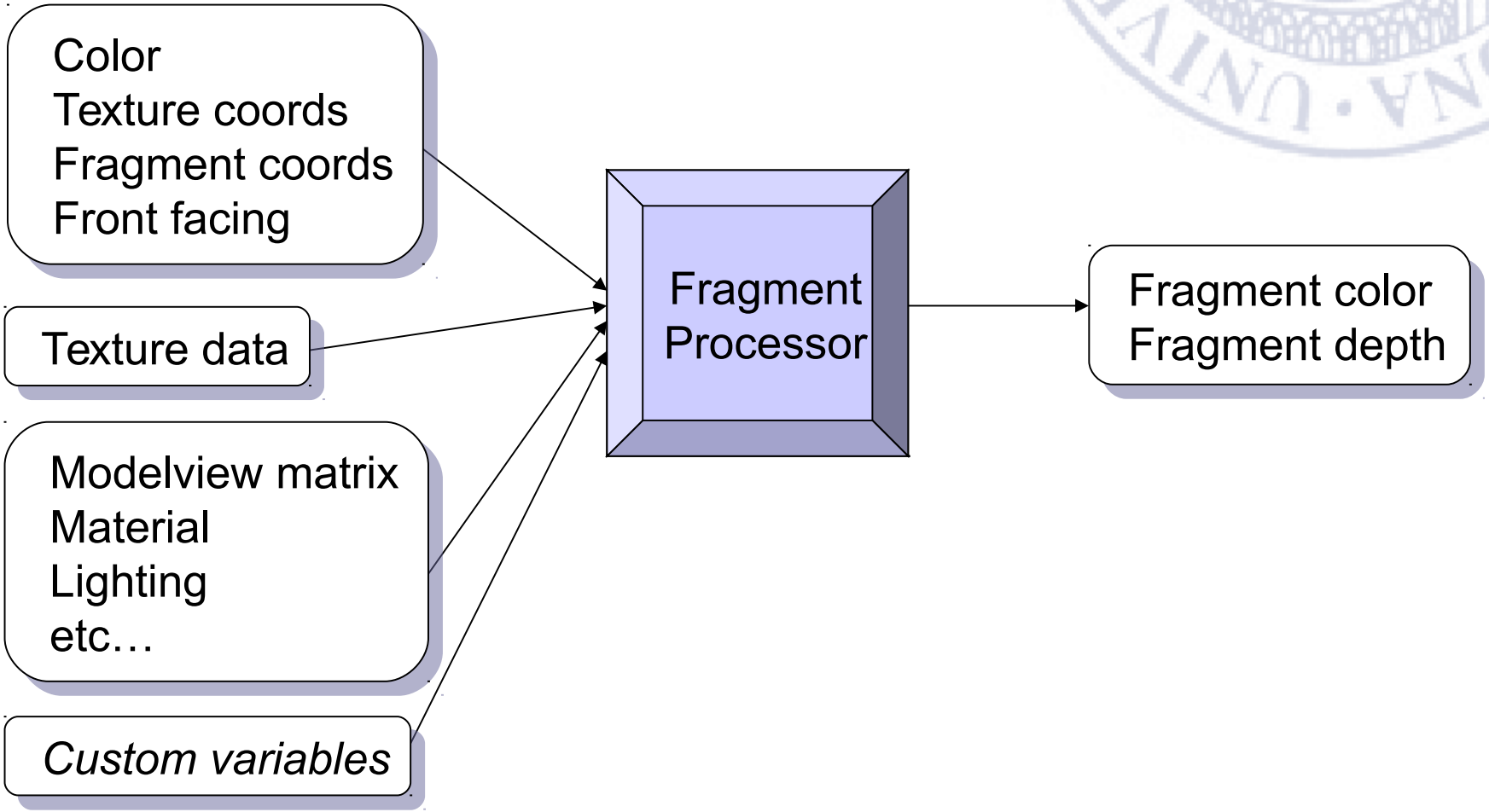
```
in                  vec4 vertex;        // vertex attribute
```

- Due modi per passare dati a questa variabile:
  - Se solo array vertici (Vertex Array),
  - Oppure un VBO (Vertex Buffer Object) contenente tutti gli attributi
- Passi
  - Ottenere la locazione
  - Abilitare l'array di attributi
  - Settare puntatore all'array
  - Disegnare e disabilitare l'array

Vertex shader input e output



Fragment shader input e output



Nota

- in OpenGL, tutti I tipi di oggetti (non in senso OOP) come buffer, textures, sono gestiti allo stesso modo
- Creazione e inizializzazione:
  - Prima di tutto di crea un handle (o un “nome”) per l'oggetto
  - Poi si collega (bind) l'oggetto, per renderlo “current”.
  - Si passano i dati OpenGL (una volta per tutte).
  - Si scollega l'oggetto se non usato
- Utilizzo in rendering
  - Si collega (bind) l'oggetto, per renderlo “current”.
  - Si usa l'oggetto
  - Si scollega l'oggetto se non usato
- Alla fine (quando non serve più) si distrugge

Tutorial

- Cercheremo di introdurre la programmazione moderna OpenGL (pipeline programmabile)
- Programmi in C++ ma manterremo il codice sostanzialmente similare al C
- Compileremo in modo semplice, da linea di comando o con semplici makefile (non c'è molto da debuggare), ma siete liberi di usare ambienti di sviluppo diversi
  - Ma non se vi complica la vita
- Esercizi da scaricare sul sito di e-learning come archivi autoconsistenti
  - Verranno aggiunti moduli/esercizi incrementalmente

La cartella di lavoro

- Contiene le librerie da usare e i relativi include per ogni piattaforma
- Linux, 32 bit, mac windows
- Linux: indicare path per l'esecuzione
  - export LD_LIBRARY_PATH=lib/lin

  - NB sulle macchine di Lab Delta le schede grafiche non supportano la versione di openGL del tutorial
  - Occorre usare sw:
  - export LIBGL_ALWAYS_SOFTWARE=1

Compilazione

Istruzioni all'interno dei file:

- Linux :
  - c++ -o app e01.cpp -I ./include -L./lib/lin -Wl,-rpath,./lib/lin/ -lglfw -lGL
- linux 32 :
  - c++ -o app e01.cpp -I ./include -L./lib/lin32 -Wl,-rpath,./lib/lin32/ -lglfw -lGL
- Per Linux anche Makefile semplice: make nome_esercizio
- Windows :
  - CL /MD /Feapp /D "_DEBUG" /Iinclude e01.cpp /link /LIBPATH:lib\win /NODEFAULTLIB:MSVCRTD
- osx :
  - c++ -o app e01.cpp -I ./include -L ./lib/mac -lglfw3 -framework Cocoa -framework OpenGL -framework IOKit -framework CoreVideo

GLFW

- OpenGL si occupa solo del rendering 2D e 3D
- Delle finestre si occupano i differenti window handler
  - opening, closing, and displaying windows
  - user interface components
  - event management
- Ogni sistema ha le sue interfacce a OpenGL
- Si possono usare librerie varie da molto semplici a complicate
- Qui useremo GLFW http://www.glfw.org/
  - Multipiattaforma
  - Facile creazione contesto openGL
  - Supporto per OpenGL 3.2+ e OpenGL ES

Iniziamo

- Occorre includere glfw3.h

```
#include <GLFW/glfw3.h>
```

- Contiene costanti e prototipi di funzione dell' API GLFW. Include l'header OpenGL, che non va quindi incluso direttamente
- Non includere neppure header platform-specific, a meno che non vadano usati direttamente. Nel caso includerli prima
- Dalla versione 3.0, l'header GLU glu.h non è più incluso automaticamente, se si vuole inserire, GLFW_INCLUDE_GLU prima dell'header GLFW

```
#define GLFW_INCLUDE_GLU
#include <GLFW/glfw3.h>
```

Configurare la finestra

- Per prima cosa si inizializza la libreria con glfwInit

```
if ( !glfwInit() )
    exit( EXIT_FAILURE );
```

- Finito l'uso, in genere fine programma, chiamare `glfwTerminate();`
- Distrugge tutte le finestre e libera tutte le risorse impegnate da glfw

Finestra e contesto

- Per creare finestra e contesto si usa glfwCreateWindow,che ritorna un puntatore alla finestra

```
GLFWwindow* window = glfwCreateWindow(640,
480, "My Title", NULL, NULL);
```

- Se fallisce ritorna NULL

```
if (!window)
{
    glfwTerminate();
    exit(EXIT_FAILURE);
}
```

Finestra e contesto (2)

- Il puntatore è passato alle funzioni che riguardano le finestre e associato agli eventi di input permettendo di selezionare la finestra che li genera
- Per creare la finestra full screen si seleziona il monitor. In molti casi va bene il primary monitor, che viene dato da glfwGetPrimaryMonitor.

```
GLFWwindow* window = glfwCreateWindow(640, 480,
"My Title", glfwGetPrimaryMonitor(), NULL);
```

- La risoluzione viene settata a quella consentita più vicina alla richiesta
- Per distruggere le finestre si usa

```
glfwDestroyWindow(window);
```

GL context

- Per usare OpenGL occorre avere un OpenGL context attivo
- Si fa con glfwMakeContextCurrent. Rimane tale finché non si cambia context o la finestra che lo possiede viene distrutta

```
glfwMakeContextCurrent(window);
```

- Ogni finestra ha un flag per indicare che dovrebbe essere chiusa si setta a 1 quando si agisce sul comando della finestra o con combinazione di tasti (Alt+F4). Non si chiude direttamente ma si deve controllare e chiudere o avvisare
- Notifica con glfwSetWindowCloseCallback.
- Si può settare il flag come si vuole con glfwSetWindowShouldClose.

Eventi di input

- Le finestre possono avere vari callback per ricevere vari tipi di eventi. Per ricevere eventi da tastiera si usa glfwSetKeyCallback

```
static void key_callback(GLFWwindow* window,
int key, int scancode, int action, int mods)
{
if (key==GLFW_KEY_ESCAPE && action==GLFW_PRESS)
      glfwSetWindowShouldClose(window,GL_TRUE);
}
```

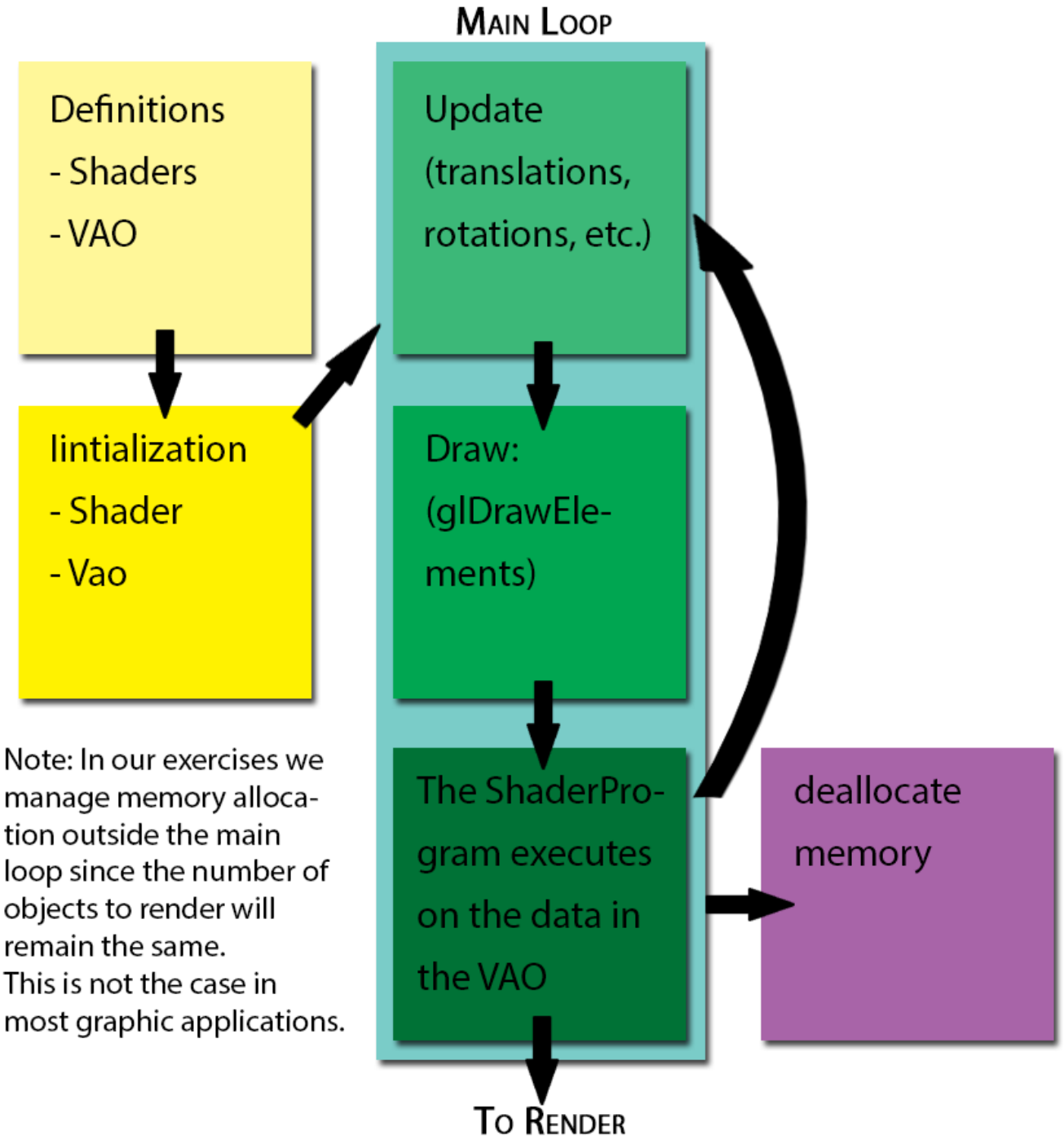
- Occorre gestire i callback con le funzioni opportune

Funzioni glfw utilizzate

- glfwGetFramebufferSize(window, &width, &height);
  - Ricava dimensioni frame buffer
- glfwSwapBuffers(window);
  - Scambia i buffer per la visualizzazione
- GlfwPollEvents();
  - Controlla gli eventi di input

Struttura programma

- main():
  - Creazione finestra
  - Inizializzazione
    - Inizializzazione librerie caricamento file di configurazione
    - Allocazione memoria, preprocessing, ...
    - Alcuni usano funzione init() per farlo
  - Start main window-loop
    - Tipicamente funzioni draw() o render() e update()
  - Liberare le risorse
    - A volte anche questo demandato a funzione
  - Uscita dall'applicazione



Note: In our exercises we manage memory allocation outside the main loop since the number of objects to render will remain the same.
This is not the case in most graphic applications.

Esempio

- e01.cpp
- Commentiamo il codice
- Strutturato in funzioni per visualizzare meglio la struttura dell'applicazione. Cominciamo dal main()

Struttura programma

- main():
  - Creazione finestra
  - Inizializzazione
    - Inizializzazione librerie caricamento file di configurazione
    - Allocazione memoria, preprocessing, …
    - Alcuni usano funzione init() per farlo
  - Start main window-loop
    - Tipicamente funzioni draw() o render() e update()
  - Liberare le risorse
    - A volte anche questo demandato a funzione
  - Uscita dall'applicazione

```
int main(int argc, char const *argv[])
{
    GLFWwindow* window;
    glfwSetErrorCallback(error_callback);

    if(!glfwInit())
        return EXIT_FAILURE;

// parte omessa

    window = glfwCreateWindow(800, 800, "cg-lab", NULL,
NULL);
    if(!window)
    {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    glfwMakeContextCurrent(window);
```

Struttura programma

- main():
  - Creazione finestra
  - Inizializzazione
    - Inizializzazione librerie caricamento file di configurazione
    - Allocazione memoria, preprocessing, ...
    - Alcuni usano funzione init() per farlo
  - Start main window-loop
    - Tipicamente funzioni draw() o render() e update()
  - Liberare le risorse
    - A volte anche questo demandato a funzione
  - Uscita dall'applicazione

```
//parte omessa
    glfwSetKeyCallback(window, key_callback);

    initialize_shader(); check(__LINE__);
    initialize_vao(); check(__LINE__);

    while(!glfwWindowShouldClose(window))
    {
        draw(window); check(__LINE__);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    destroy_vao(); check(__LINE__);
    destroy_shader(); check(__LINE__);

    glfwDestroyWindow(window);
    glfwTerminate();
    return EXIT_SUCCESS;
```

```
void initialize_shader()
{
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexSource, NULL);
    glCompileShader(vertexShader);

    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
    glCompileShader(fragmentShader);

    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);

    glBindFragDataLocation(shaderProgram, 0, "outColor");
    glLinkProgram(shaderProgram);

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);}
```

```
void initialize_vao()
{
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    glBufferData(GL_ARRAY_BUFFER,
 sizeof(vertices), vertices, GL_STATIC_DRAW);
// sizeof(vertices) = sizeof(GLfloat) * 4
(vertici) * 2 (coordinate x,y)
    glGenBuffers(1, &ibo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);

    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
        sizeof(elements), elements, GL_STATIC_DRAW);

                //sizeof(elements) = sizeof(GLuint) * 2 (triangoli) * 3 (vertici per
                triangolo
    GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
    glEnableVertexAttribArray(posAttrib);
    glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
        2 * sizeof(GLfloat), 0);
}
```

```
void draw(GLFWwindow* window)
{
    int width, height;
    glfwGetFramebufferSize(window, &width, &height);

    // le dimensioni sono in pixel, passiamo quelle
    //del framebuffer perche' la posizione calcolata a schermo dal
    //sistema operativo potrebbe essere diversa (retina)
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT);

    // prima di disegnare dobbiamo avere sempre
    //UN program e UN vao attivo
    glUseProgram(shaderProgram);
    glBindVertexArray(vao);

    // GL_TRIANGLES =  l'elements buffer verra' letto a triplette
    //e disegnate come triangoli indipendenti
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}
```

```
static void error_callback(int error, const char* description)
{
    std::cerr << description << std::endl;
}
/*   funzione richiamata da glfw ogni volta che
viene rilevata la pressione o il rilascio di un tasto (della tastiera)*/

static void key_callback(GLFWwindow* window, int key,
     int scancode, int action, int mods)
{
    /* alla pressione del tasto ESC cambio il flag per segnalare
    che vogliamo terminare l'esecuzione dell'applicazione
    nel main glfwPollEvents viene richiamata al termine
    del while quindi l'uscita sara' immediata, diversamente se la
    richiamassimo prima di glfwSwapBuffers potrebbe esserci
    dell'attesa dovuta alla sincronizzazione di opengl stesso*/
     if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
          glfwSetWindowShouldClose(window, GL_TRUE);
}
```

```
// Il codice degli shader viene qui inserioto in un vettore char
// si può poi caricare da file come a p.26

//Shader sources
/*passiamo alla gpu la posizione in screen-space quindi non c'e‘
nessuna trasformazione da applicare */
const GLchar* vertexSource =
```

Non mettiamo trasformazioni geometriche! Si disegna direttaente sullo screen space.
Solo c'è la mappatura di default tra [-1, 1]
Trasformo poi in 3D aggiungendo lo 0

```
    "in vec2 position;"
    "void main() {"
    "    gl_Position = vec4(position, 0.0, 1.0);"
    "}";

/* applichiamo un colore uniforme (bianco) a tutti i pixel*/
const GLchar* fragmentSource =

    "out vec4 outColor;"
    "void main() {"
    "    outColor = vec4(1.0);"
    "}";
```

Nota: non usa input
tutti i pixel creati dai frammenti mappati saranno bianchi

Definisce struttura dati per il modello. Qui siamo in 2D non mettiamo la z

```
const GLfloat vertices[] = {
    -0.5f,  0.5f, // Top-left
     0.5f,  0.5f, // Top-right
     0.5f, -0.5f, // Bottom-right
    -0.5f, -0.5f, // Bottom-left
};
const GLuint elements[] = {
    0, 1, 2,
    2, 3, 0
};
```

Coordinate vertici

Connettività (triangoli)

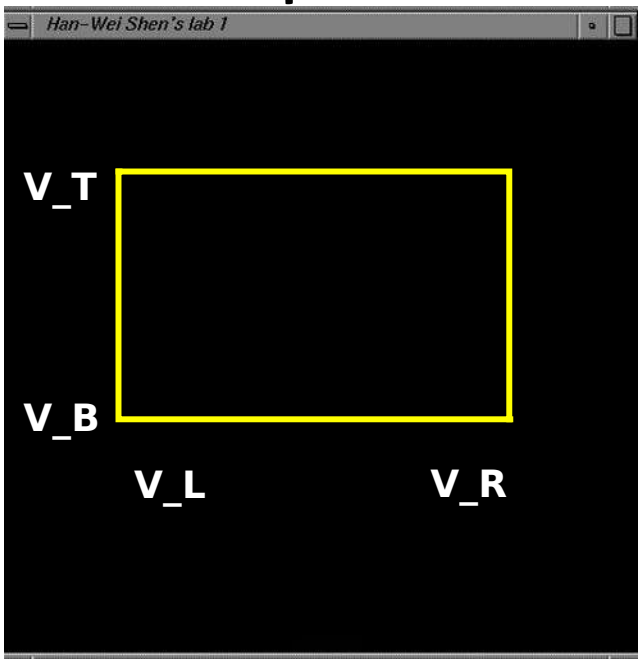
Note

- I possibili valori per glDrawElements sono
  - GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, and GL_TRIANGLES

- GlClearColor(R,G,B,A) cambia il colore con cui viene ripulito il frame buffer

Viewport

- La regione rettangolare dello schermo dove si mappano le coordinate proiettate
- Definita nel sistema di riferimento della finestra gestita dal sistema operativo




```
viewport(int  left, int  bottom,
         int  (right-left),
         int (top-bottom));
```

Esercizio

- Partire dal codice e02.cpp
- Aggiungere le parti mancanti per creare un quadrato simile al precedente, ma coi vertici colorati in rosso, verde blu e bianco
- Come viene interpolato il colore dei vertici sui pixel?
  - Cercare di spiegare.
  - Fare sì che all'avvio lo sfondo sia blu e schiacciando il tasto R lo sfondo diventi rosso

Riferimenti

- http://www.opengl.org
- http://www.khronos.org/opengl/
- http://www.glfw.org/
- http://antongerdelan.net/opengl
- http://www.opengl-tutorial.org/
- http://www.arcsynthesis.org/gltut/
- http://glm.g-truc.net/0.9.5/index.html