Libreria Grafica

WebGL

https://www.khronos.org/webgl/

Che cosa è OpenGL ?

OpenGL (Open Graphics Library) è una libreria per creare applicazioni interattive, per gestire oggetti geometrici 3D ed immagini, per ottenere grafica ad alte prestazioni; inizialmente utilizzabile solo da codice C/C++

E' una libreria grafica indipendente dal sistema operativo e dal Window System, nel senso che permette il rendering, ma non specifica come gestire finestre e ricevere eventi dal sistema.

Ogni **Window System** che supporta OpenGL offre quindi chiamate aggiuntive per integrare OpenGL nella gestione di finestre, colormap ed altre caratteristiche.

Un po' di Storia

- Inizialmente si chiamava **IRIS GL,** era sviluppata da **Silicon Graphics** ed era proprietaria (1980)

- Dal 1992 OpenGL **ARB** (**A**rchitecture **R**eview **B**oard)
  - mantiene e aggiorna le *specifiche*
  - una compagnia, un voto
  - inoltre c'erano le *estensioni* private: ***ATI*** e ***n*VIDIA**

- Dal 2007 gestita da OpenGL Khronos Group
  - ultima versione rilasciata: 4.6 (agosto 2017)

API (Application Programming Interface)

Librerie per interfacciarsi con **OpenGL (GL)**:

- **GLX** in X Window System
- **CGL** in Apple Macintosh
- **WGL** in Microsoft Windows
- **EGL** per OpenGL ES su mobile ed embedded device

- **GLU** (OpenGL Utility Library)
  - Inclusa nella **GL** definisce funzioni di utilità e altre primitive grafiche (curve e superfici NURBS)
- **GLEW** (OpenGL Extension Wrangler Library)
  - semplifica l'accesso alle funzioni OpenGL ed il lavoro con le OpenGL extension

Una appliccazione utile su Windows e Mac: **GLview** (OpenGL Extension Viewer)

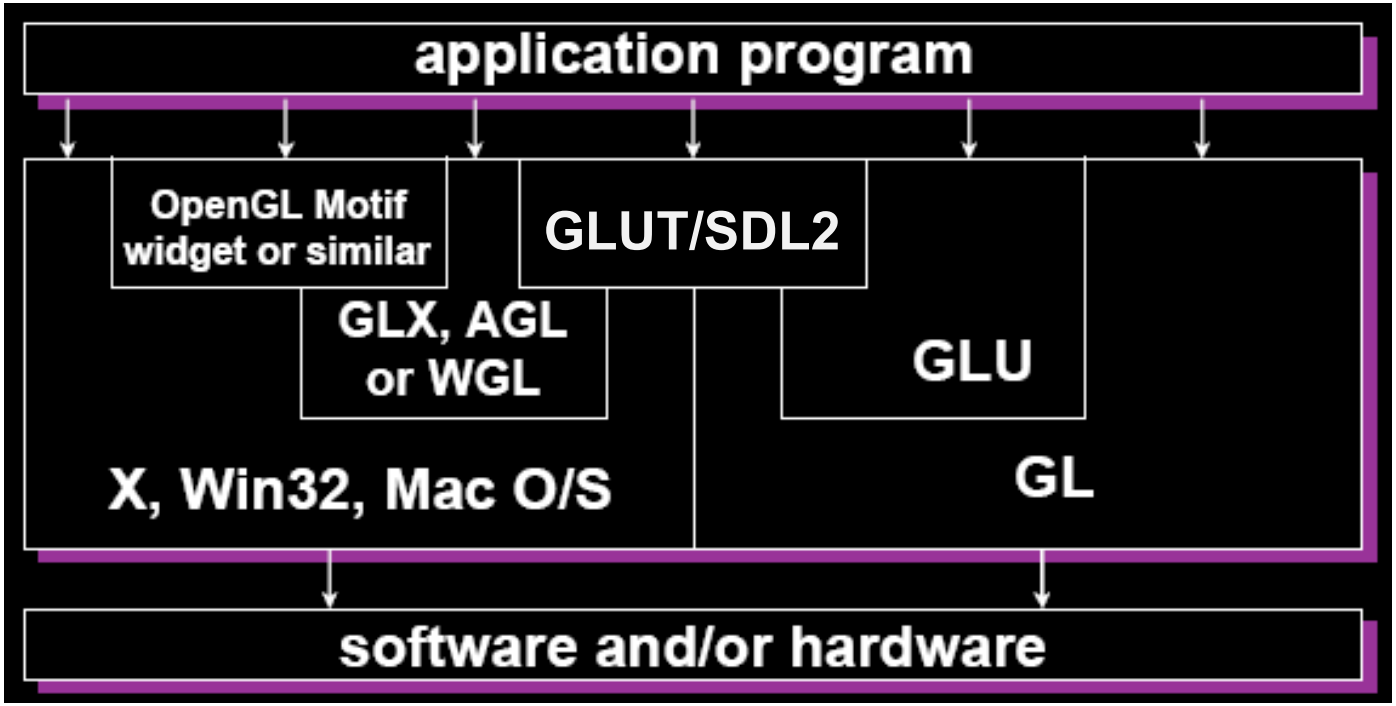
http://realtech-vr.com/admin/glview

API (Application Programming Interface)

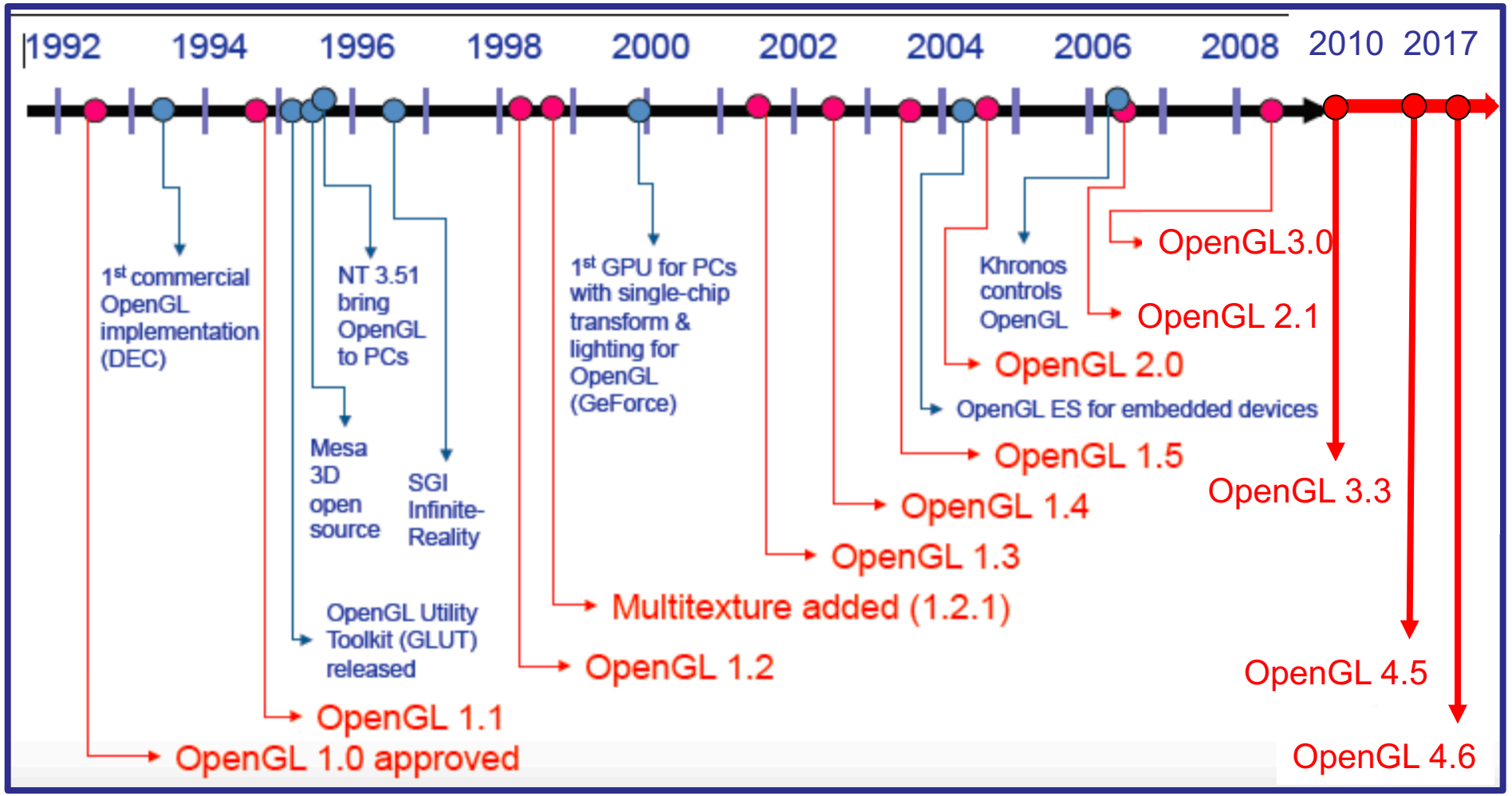
- **GLUT** (OpenGL Utility Toolkit) (più precisamente freeglut)
  - Funzioni per creare semplici interfacce utente, gestire eventi (keyboard e mouse); (interfaccia con il Window System)

- **SDL** (Simple Directmedia Layer)
  - Funzioni per gestire eventi (keyboard e mouse) e molto altro; (interfaccia con il Window System)

API (Application Programming Interface)

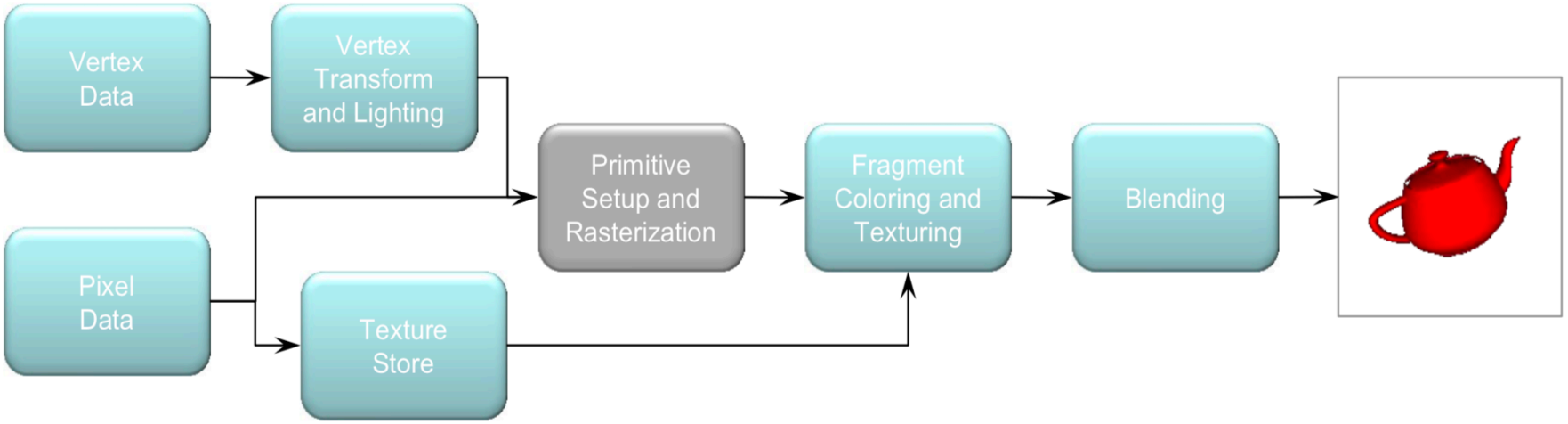


Evoluzione di OpenGL



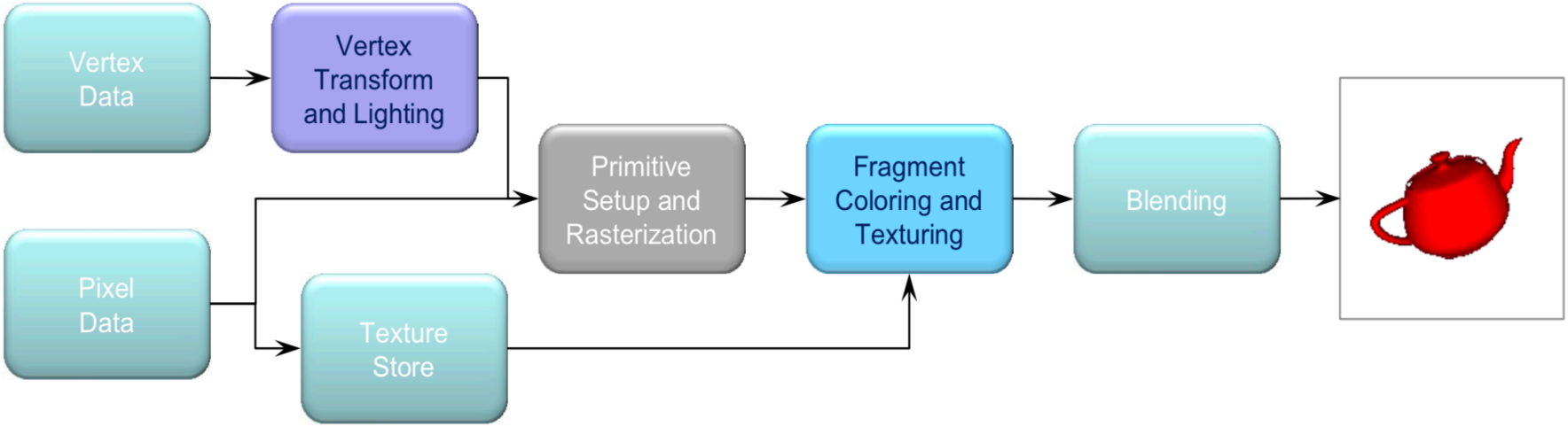
OpenGL

OpenGL 1.0 fu rilasciata l'1 luglio 1994; la sua pipeline era interamente fissa. La pipeline ha successivamente avuto delle evoluzioni, ma rimanendo fissa dalla versione 1.1 fino alla 2.0 (settembre 2004).



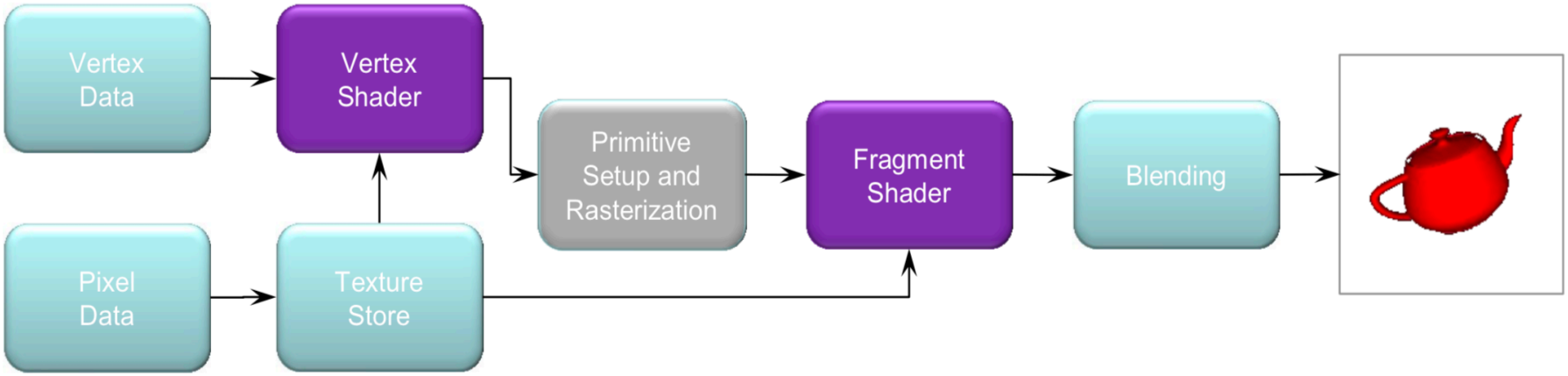
OpenGL

OpenGL 2.0 aggiunse la programmabilità degli shaders.
Comunque, la pipeline fissa era ancora disponibile.



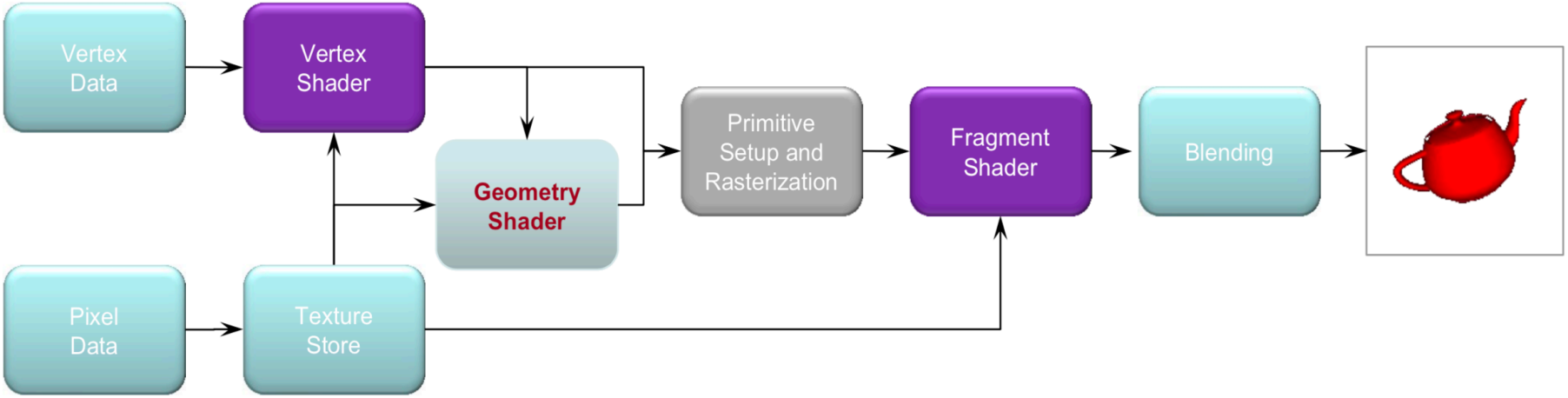
OpenGL

OpenGL 3.1 rimosse la pipeline fissa; i programmi dovevano usare gli shaders.



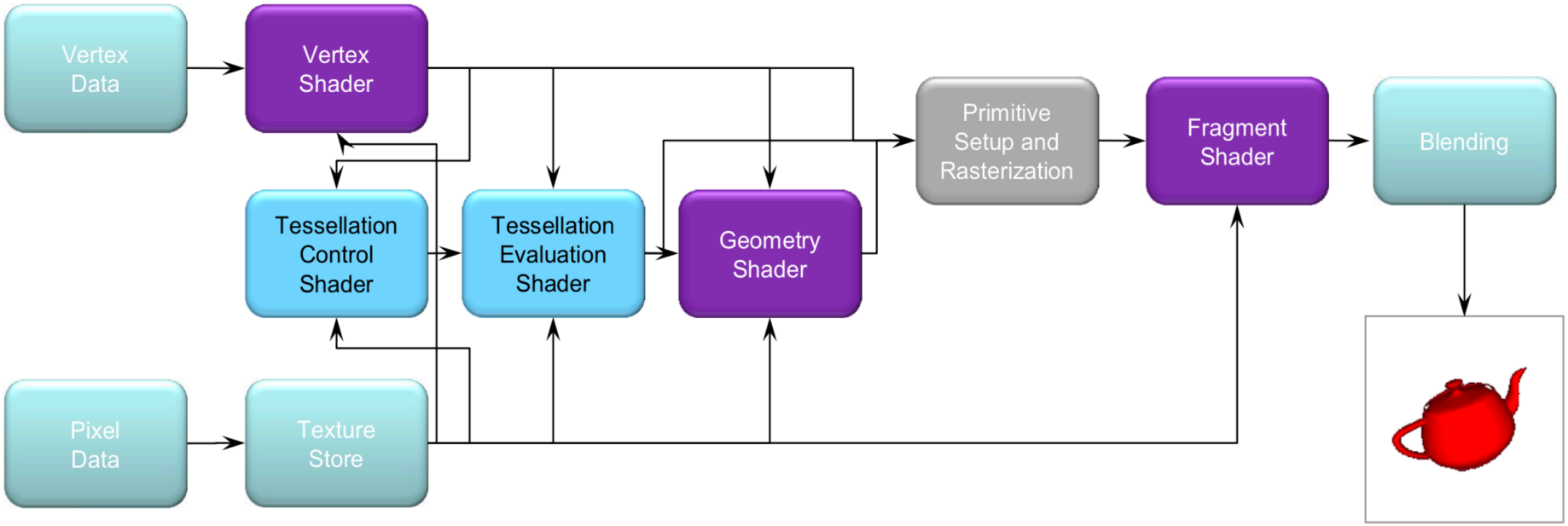
OpenGL

OpenGL 3.2 (rilasciata il 3 agosto 2009) aggiunge una ulteriore fase di shading, il Geometry Shader (permette di modificare la geometria all'interno della pipeline).



OpenGL

OpenGL 4.1 (rilasciata il 25 luglio 2010) include ulteriore fasi di shaders, il Tessellation-Controll e il Tessellation-Evaluation shader.
L'ultima versione è la 4.6.



OpenGL (riassumendo)

- Fixed-Function Pipeline

➢Fino alla OpenGL 1.5

➢Dalla OpenGL 2.0 alle 3.1 massima compatibilità fra i due modi, con le fixed function emulate da shaders

➢Dalla OpenGL 3.2 alcune fixed function non sono più emulate

- Programmable Pipeline

➢Dalla OpenGL 2.0, Pixel e Fragment shaders

➢OpenGL Shading Language (GLSL)

Mobile/Web API

- **OpenGL ES 3.0**
  - progettatta per embedded device
  - basata su OpenGL 3.1
  - basata su shaders
- **Web GL 1.0 e 2.0**
  - implementazioni JavaScript di OpenGL ES 2.0 ed ES 3.0
  - presenti sui più recenti browser
- **OpenGL ES SL 3** (OpenGL Embedded System Shading Language)
  - linguaggio di programmazione rilasciato dalla versione 2.0 di OpenGL in poi
  - API appositamente studiate per sistemi embedded come quelli presenti sui dispositivi mobili, telefoni e tablet
  - in OpenGL, WebGL e OpenGL ES, si utilizza OpenGLSL per programmare gli shaders

Web API (WebGL Application Programming Interface)

GLSL ES version	OpenGL ES version	WebGL version	Based on GLSL version	Date	Shader Preprocessor
1.00.17	2.0	1.0	1.20	12 / 5 / 2009	#version 100
3.00.6	3.0	2.0	3.30	29 / 1 / 2016	#version 300 es

KHRONOS GROUP

CONNECTING SOFTWARE TO SILICON

https://www.khronos.org/webgl/

Perché WebGL ?

-**Programmazione JavaScript**: le applicazioni WebGL sono scritte in JavaScript.

-**Supporto Mobile**: WebGL supporta anche i browser su Mobile come iOS Safari e Chrome per Android.

-**Open source:** WebGL è open source.

-**No compilazione**: poiché le applicazioni WebGL sono in JavaScript, non è necessario compilare

-**Gestione della memoria**: non è necessario allocare esplicitamente la memoria, questa è automatica.

-**Facile da usare**: poiché WebGL è integrato in HTML 5, non sono necessarie installazione aggiuntive. Per scrivere un'applicazione WebGL, tutto ciò che serve è un editor di testo e un web browser.

HTML5+canvas+contesto webgl

Abbiamo già visto l'elemento canvas e il contesto 2d; ora vediamo il contesto WebGL.

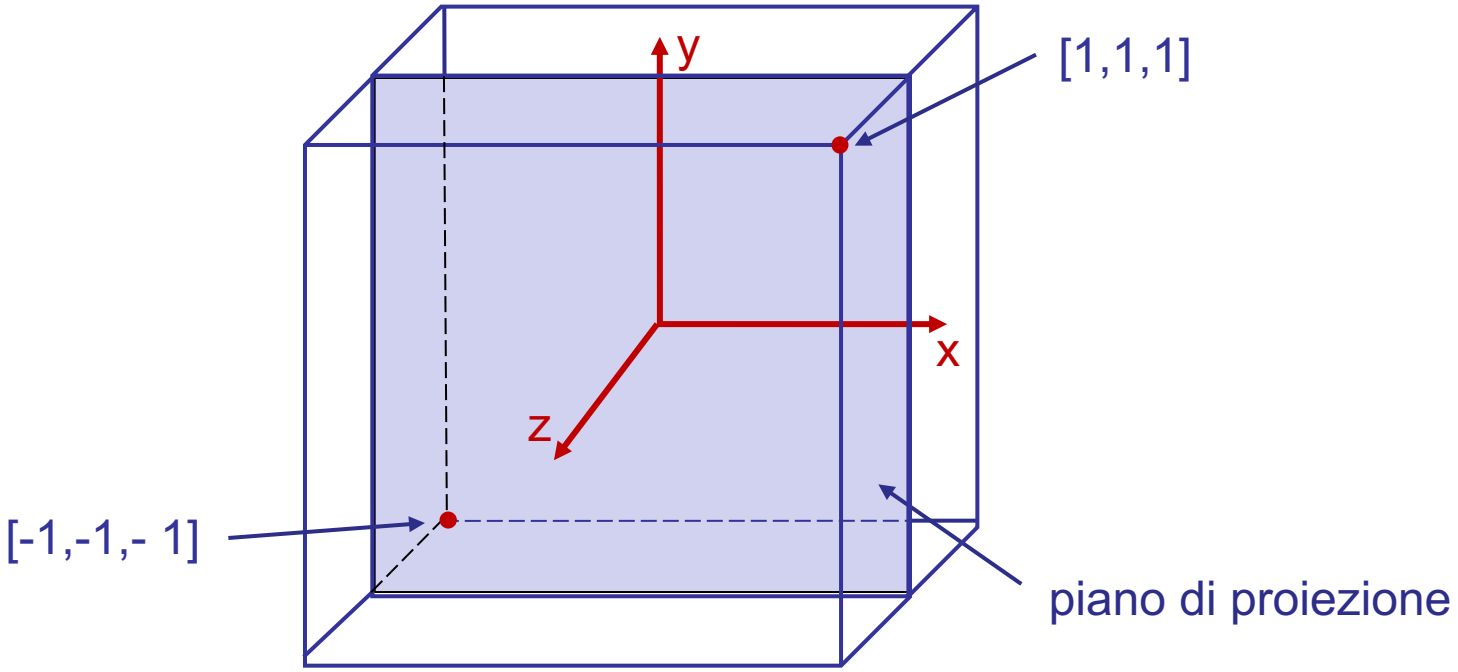
Per creare un contesto di rendering WebGL sull'elemento canvas, è necessario passare la stringa experimental-webgl (od anche solo webgl), anziché 2d, al metodo canvas.getContext ().

```
<!DOCTYPE html>
<html>
   <canvas id='my_canvas'></canvas>
   <script>
     var canvas = document.getElementById('my_canvas');
     var gl = canvas.getContext('webgl');
     gl.clearColor(0.9,0.9,0.8,1);
     gl.clear(gl.COLOR_BUFFER_BIT);
   </script>
</html>
```

Sistema di Coordinate

In WebGL il sistema 3D è quello dell'Osservatore, ma destrorso, dove quindi l'asse z indica la profondità, ma in senso opposto (l'osservatore guarda nel verso negativo dell'asse z).
Le coordinate sono limitate fra [-1,-1,-1] e [1,1,1].

WebGL non visualizzerà nulla se fuori da questo cubo/frustum.



Grafica WebGL

Dopo aver ottenuto il contesto WebGL dell'elemento canvas, è possibile iniziare a disegnare elementi grafici utilizzando le API WebGL.
Come abbiamo visto, una geometria mesh 3D si definisce tramite una lista di vertici ed una lista di facce; per es. la seguente è una faccia triangolare 3D

```
var vertices = [ -1,-1,-1,  1,-1,-1,  1, 1,-1 ];
var indices = [ 0, 1, 2 ];
```

Buffer Object

I buffer sono le aree di memoria WebGL, su GPU, che contengono i dati. Esistono vari buffer, i più importanti sono il Vertex Buffer Objcet (VBO) e l'Index Buffer Object (IBO) e vengono utilizzati per contenere la geometria del modello.

Dopo aver definito/memorizzato i vertici in array JavaScript, li passiamo alla GPU memorizzandoli in questi Buffer Object di WebGL.

Grafica WebGL

Per disegnare oggetti 2D o 3D, l'API WebGL fornisce due metodi:

drawArrays( ) e drawElements( )

Questi due metodi accettano un parametro chiamato mode mediante il quale è possibile selezionare l'oggetto che si desidera disegnare; le possibilità sono limitate a punti, linee e triangoli.

```
drawArrays(gl.TRIANGLES, … , …);
```

Invocando da JavaScript uno di questi metodi, inizia l'esecuzione della pipe-line grafica o di rendering su GPU che produrrà un'immagine su schermo.

Shader Programs

Poiché WebGL utilizza ES SL (Embedded System Shader Language) che gira su GPU, scriveremo i programmi per disegnare, usando shader programs.

Gli shader sono i programmi per GPU; il linguaggio utilizzato è GLSL. In questi shader programs, definiamo esattamente come i vertici, le trasformazioni e la camera (oltre, come vedremo, i materiali e le luci) interagiscono tra loro per creare un'immagine.

In breve, è un codice che implementa algoritmi per ottenere i pixel immagini di una mesh.

Esistono due tipi di shader programs:

Vertex Shader e Fragment Shader

Vertex Shader

Vertext shader è il codice del programma chiamato su ogni vertice.

Viene utilizzato per trasformare (spostare) da una posizione ad un'altra la geometria, vertice per vertice.

Gestisce i dati di ciascun vertice (dati per vertice) come le coordinate dei vertici, le normali, i colori e le coordinate texture.

Nel codice ES GL del vertex shader, i programmatori devono definire gli attribute per gestire i dati. Questi attribute si riferiscono ad un Vertex Buffer Object.

Un Vertex Shader può:
Trasformare vertici, trasformare e normalizzare normali, generare coordinate texture, trasformare coordinate texture, gestire l'illuminazione, applicare materiali.

Fragment Shader

Una mesh è formata da più triangoli e la superficie di ciascuno dei triangoli è nota come un fragment.

Fragment shader è il codice che viene eseguito su ogni pixel di ogni fragment.

Viene scritto per calcolare e definire il colore dei singoli pixel.

Un Fragment Shader può:
Eseguire operazioni sui valori interpolati dei pixel, accedere alle texture, applicare texture, sommare colori, ecc.

Variabili OpenGL ES SL

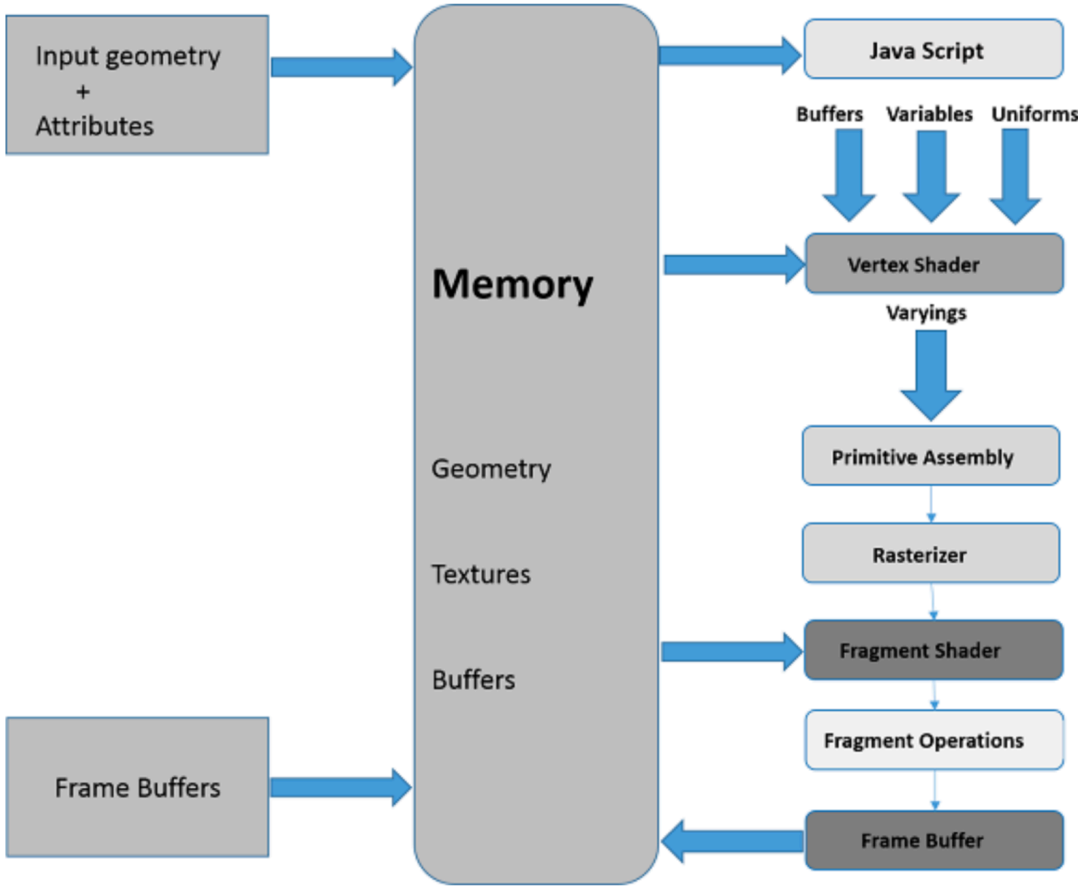
Per gestire i dati negli shader programs, OpenGL ES SL fornisce tre tipi di variabili.

-Attributes: sono le variabili che contengono i valori di input del programma Vertex Shader. Indicano il Vetex Buffer Object che contiene i dati dei vertici. Ogni volta che viene richiamato il vertex shader, i valori attributes variano.

-Uniforms: sono le variabili che contengono i dati di input comuni per Vertex Shader e Fragment Shader, come la posizione della luce, le coordinate texture ecc.

-Varyings: sono le variabili utilizzate per passare i dati dal Vertex Shader al Fragment Shader.

Pipeline Grafica di WebGL



Nelle slide successive vedremo il ruolo di ogni passaggio nella pipeline di figura (i titoli delle slide seguono la pipeline a destra)

JavaScript

Durante lo sviluppo di applicazioni WebGL, si scrive il codice nello Shading Language per comunicare con la GPU. Questo codice viene scritto all'interno di un codice JavaScript che include le seguenti azioni:

- Inizializza WebGL: JavaScript viene utilizzato per inizializzare il contesto WebGL.

- Crea array: si crea un array JavaScript per contenere i dati per la geometria.

- Buffer Objects: si creano buffer object (vertice e indice) passando array come parametri.

- Shaders: si scrivono, compilano e linkano i programmi vertex shader e fragment shader.

continua

JavaScript

- Attributes: si possono creare attribute, abilitarli e associarli con Buffer Object.

- Uniforms: si possono anche associare uniform.

- Matrice di trasformazione: si possono creare matrici di trasformazione.

Inizialmente si definiscono i dati per la geometria e si passano agli shader sotto forma di buffer.
Per la precisione le variabili attributes di OpenGL ES SL puntano ai Buffer Object, i quali vengono passati come input al Vertex Shader.

Vertex Shader

Invocando i metodi drawElements( ) e/o drawArray( ) inizia il processo di rendering:

parte il vertex shader che viene eseguito per ciascun vertice presente nel Vertex Buffer Object;

calcola la posizione di ciascun vertice di un poligono primitivo (punti, linee e triangoli) e lo memorizza nella variabile position di WebGL;

calcola anche gli altri attributi come colore, coordinate texture, ecc. che sono usualmente associati a un vertice.

Primitive Assembly e Rasterizer

Dopo aver calcolato la posizione e altri dettagli di ciascun vertice, la fase successiva è la fase di assemblaggio.

Ci sono due passaggi:

- Culling: inizialmente viene determinato l'orientamento (frontale o posteriore) del triangolo. Tutti quei triangoli con orientamento improprio che si trovano all'esterno del frustum vengono eliminati. Questo processo è chiamato culling.

- Clipping: se un triangolo è parzialmente al di fuori del frustum, la parte all'esterno viene rimossa. Questo processo è noto come clipping.

Quindi i nuovi triangoli vengono passati al rasterizzatore. Nella fase di rasterizzazione, vengono determinati i pixel dell'immagine finale della primitiva.

Fragment Shader

Il fragment shader prende:

- dati dal vertex shader in variabili varying,
- primitive dalla fase di rasterizzazione
- calcola i valori di colore per ciascun pixel tra i vertici

Il fragment shader memorizza i valori di colore di ogni pixel in ciascun fragment.

È possibile accedere a questi valori di colore durante le operazioni sui fragment.

Fragment Operations

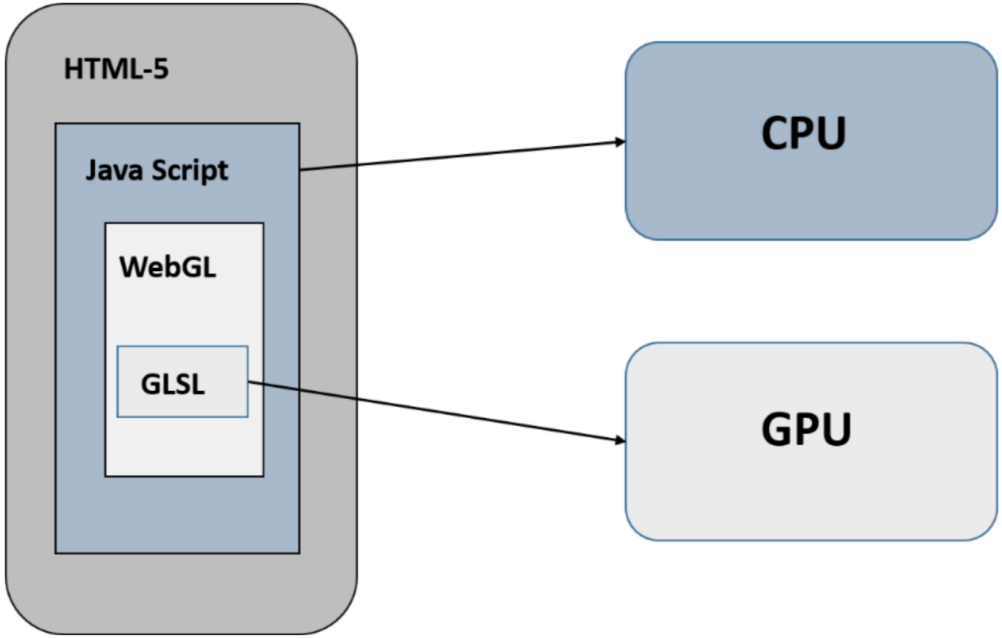
Dopo aver determinato il colore di ciascun pixel nella primitiva vengono eseguite operazioni sui fragment:

- Depth
- Color buffer blend
- Dithering

Una volta elaborati tutti i fragment, viene formata e visualizzata sullo schermo una immagine 2D.

Il frame buffer è la destinazione finale della pipeline di rendering.

Esempio di Applicazione WebGL



Pagina Web del corso, Siti, WebGL Fundamentals, State Diagram: demo Triangle

https://webglfundamentals.org/webgl/lessons/resources/webgl-state-diagram.html?exampleId=triangle#no-help

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Giulio Casciola

Dip. di Matematica

giulio.casciola at unibo.it