



# OpenGL

---



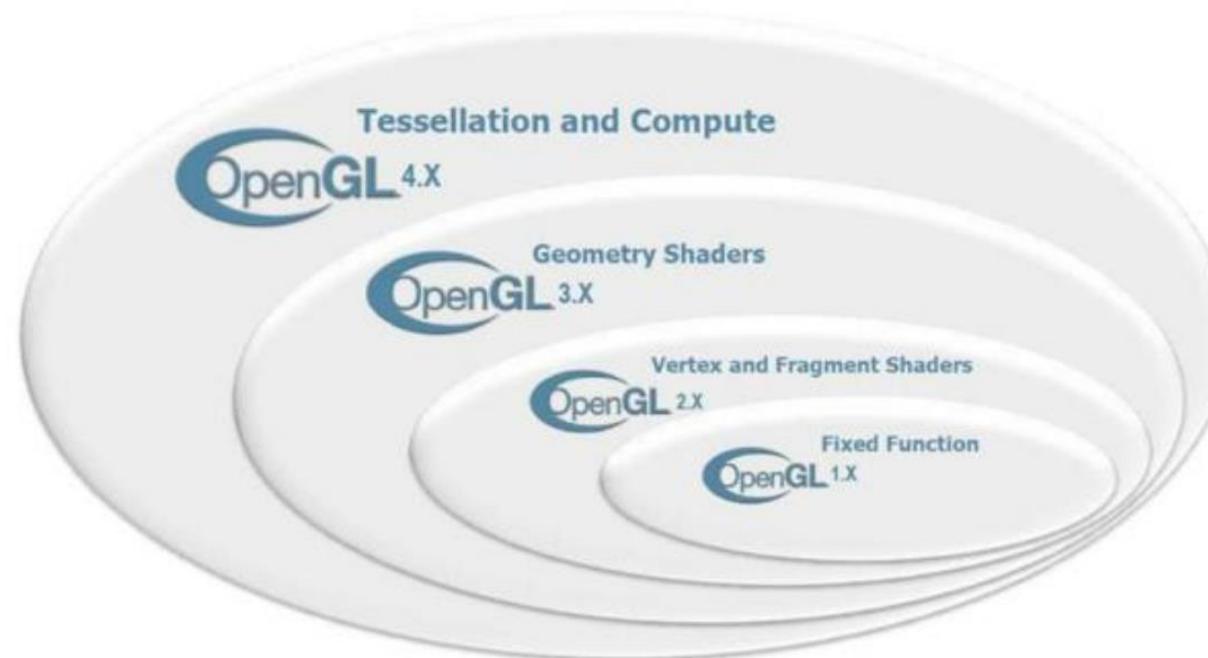
- OpenGL è una specifica che definisce un'API ( Interfaccia di programmazione applicazioni), per differenti linguaggi e sistemi operativi, che fornisce un ampio set di funzioni che possiamo usare per manipolare grafica e immagini.
  - A livello più basso OpenGL è una **specifica**, ovvero si tratta semplicemente di un **documento che descrive un insieme di funzioni ed il comportamento preciso che queste devono avere**.
  - Da questa specifica, i **produttori di hardware creano implementazioni**, ovvero librerie di funzioni **create rispettando quanto riportato sulla specifica OpenGL**, facendo uso dell'accelerazione hardware ove possibile. I produttori devono comunque superare dei test specifici per poter fregiare i loro prodotti della qualifica di implementazioni OpenGL.
  - Poiché la specifica OpenGL non ci fornisce i dettagli dell'implementazione, le versioni effettivamente sviluppate di OpenGL possono avere implementazioni diverse, a condizione che **i loro risultati siano conformi alla specifica (e quindi siano gli stessi per l'utente)**.
-



- 
- ✓ Le persone che sviluppano le attuali librerie OpenGL sono generalmente i produttori di schede grafiche. Ogni scheda grafica acquistata supporta versioni specifiche di OpenGL che sono le versioni di OpenGL sviluppate appositamente per quella scheda.
  - ✓ Quando si utilizza un sistema Apple, la libreria OpenGL è gestita da Apple stessa e sotto Linux esiste una combinazione di versioni di fornitori grafici e adattamenti di hobbisti di queste librerie.
  - ✓ Ciò significa anche che ogni volta che OpenGL mostra comportamenti strani che non dovrebbero, è molto probabile che sia colpa dei produttori di schede grafiche (o di chiunque abbia sviluppato / mantenuto la libreria).
-

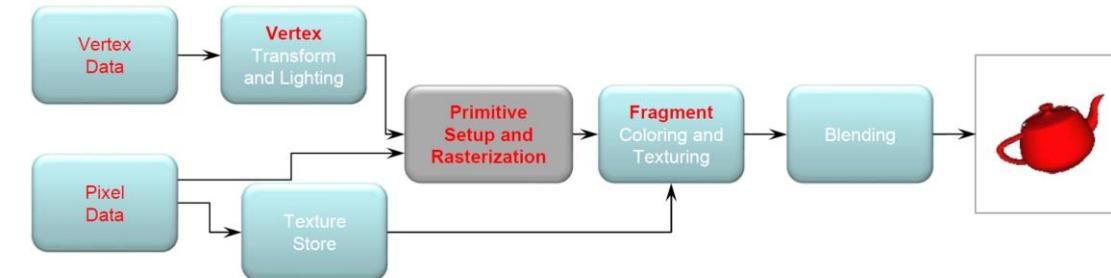


# OpenGL generation



## OpenGL v 1.0 (1994)

–Fixed Pipeline : tutte le operazioni che OpenGL supporta sono completamente definite, e un'applicazione può modificare il loro funzionamento solo cambiando un insieme di valori di ingresso (come i colori o posizioni). L'ordine delle operazioni è stato sempre lo stesso.

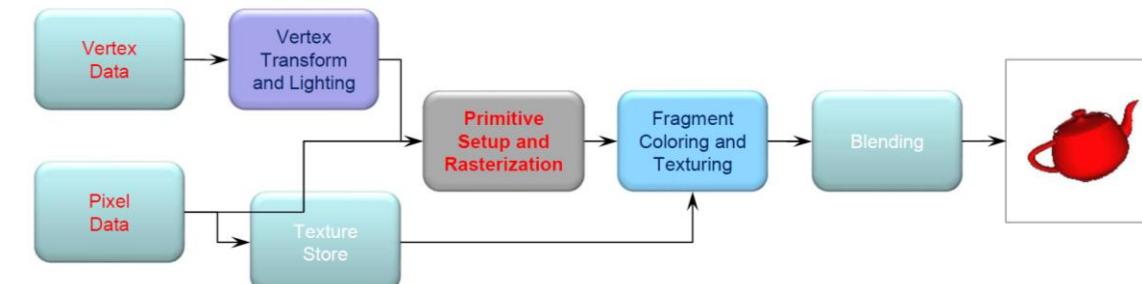


La Pipeline si evolve ma rimane sempre fissa fino alla versione 1.1 (settembre 2004)

## •OpenGL v 2.0 (2004)

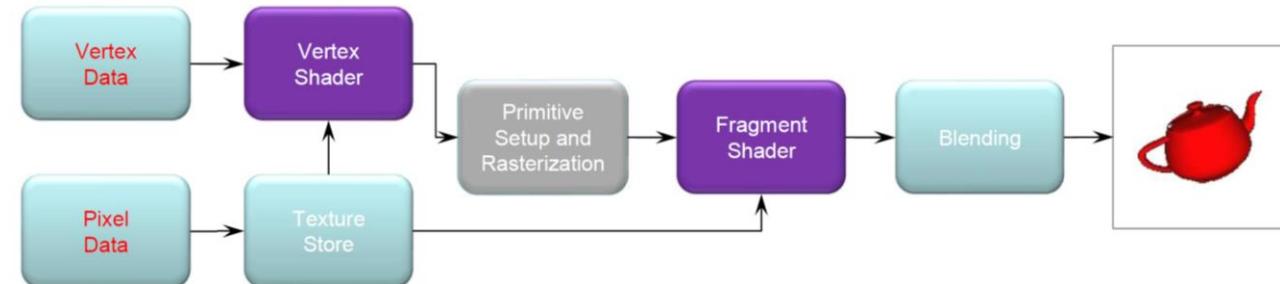
E' stato introdotto il linguaggio di shading (GLSL) che permette la programmazione di trasformazioni sui vertici (vertex shader) e lo shading di frammenti (fragment shader).

La fixed pipeline è ancora utilizzabile



## •OpenGL v 3.1 (2009)

–Non c'è più il support per le fixed function pipeline, i programmi devono usare solo shaders.



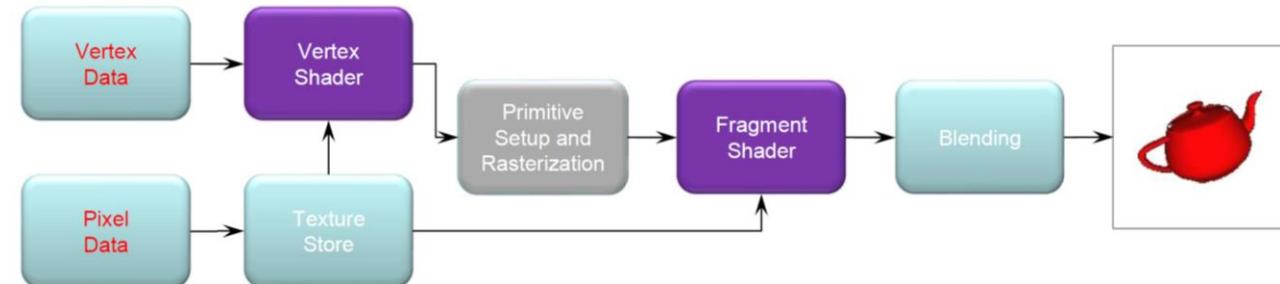
Quasi tutti i dati dei vertici sono residenti su GPU

## •OpenGL v 3.2 (2009)

- Sono stati introdotti i profile context: core e compatibility :
- Il Core profile non ha funzionalità deprecated
- Il Compatibility profile conserva le funzionalità deprecated.

## •OpenGL v 3.1 (2009)

–Non c'è più il support per le fixed function pipeline, i programmi devono usare solo shaders.



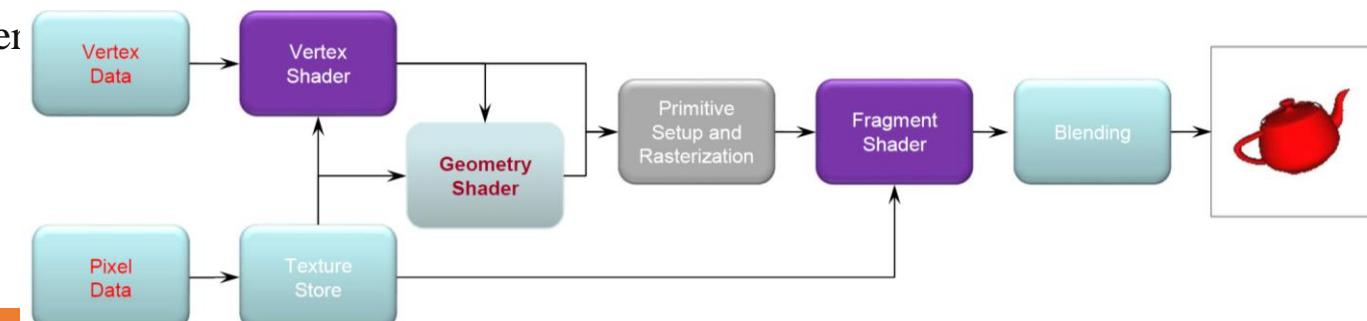
Quasi tutti i dati dei vertici sono residenti su GPU

•**OpenGL v 3.2 (2009) – Aggiunge un addizionale shading stage, il Geometry Shader,** ( strumento per espandere la geometria basica del modello, attraverso l'inclusione di un maggiori numero o diverso tipo di primitive grafiche, oltre a quelle inizialmente definite.)

–Sono stati introdotti i profile context: **core** e **compatibility** :

–Il **Core** profile non ha funzionalità deprecated (della fixed pipeline)

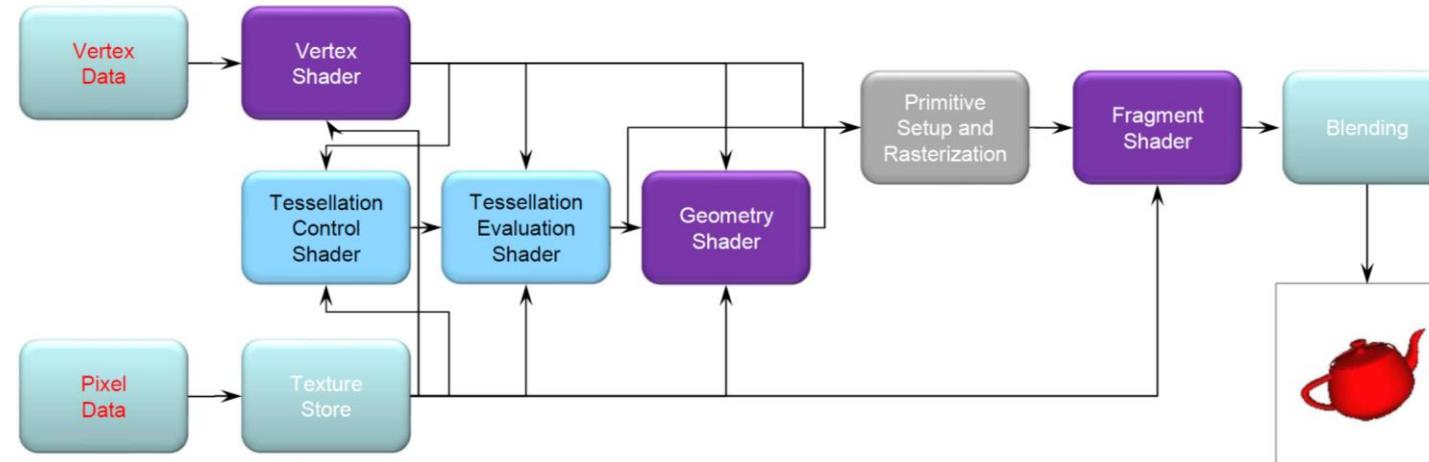
–Il **Compatibility** profile conser



# Pipeline più recenti

## OpenGL 4.1: (rilasciata il 25 luglio 2010)

Include addizionali shading stages, tessellation control e tessellation evaluation shaders. Un tessellation shader crea molta geometria a partire dalla geometria di partenza , ma tutta la nuova geometria è dello stesso tipo di quella di partenza - si possono ottenere più segmenti per una linea, più triangoli per una patch triangolare, o più isolinee o quadrilateri per una patch quadrata, ma si ottiene sempre la stessa geometria.



Versione corrente di OpenGL v 4.6 (31 Luglio 2017) e di GLSL 4.60.5

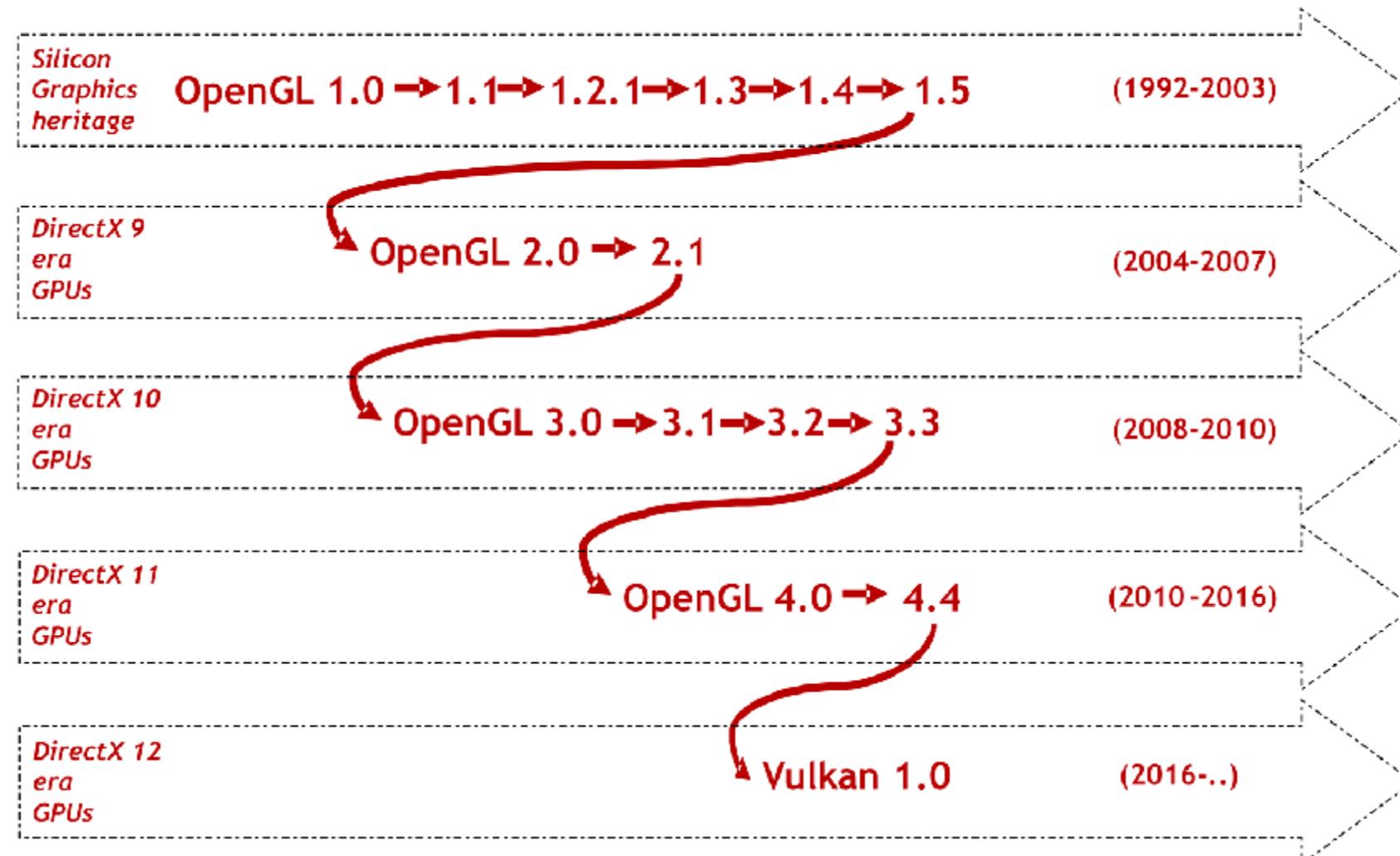
## OpenGL 4.6

Pubblicato il 31 Luglio 2017, oltre a molte estensioni e ottimizzazioni

GlView è un software per controllare la versione di OpenGL supportata da una scheda video, scaricabile dal sito <http://www.realtech-vr.com/glview/>



# OpenGL to Vulkan Progression





# Vulkan

---

Vulkan è un'interfaccia di programmazione applicativa (API) di basso livello, multi-piattaforma in 2D e 3D, annunciata la prima volta nel 2015 da Khronos Group. Inizialmente venne presentata come **"OpenGL di prossima generazione"** o semplicemente "glNext" di Khronos, ma l'uso di questi nomi è stato messo da parte una volta che è stata presentata con il nome di Vulkan. Come OpenGL, gli obiettivi di Vulkan sono le alte prestazioni per applicazioni di grafica 3D in tempo reale come i giochi e i media interattivi su tutte le piattaforme; inoltre **offre prestazioni più elevate e minore sovraccarico della CPU** come Direct3D 12 e Mantle.



Vulkan ha lo scopo di fornire una varietà di vantaggi rispetto ad altre API così come il suo predecessore OpenGL. Vulkan offre minori costi, un **controllo più diretto dell'hardware e della GPU, e un minore utilizzo della CPU, portando ai seguenti vantaggi:**

OpenGL usa il linguaggio ad alto livello GLSL per la scrittura di shader ed ogni driver OpenGL esegue in fase di esecuzione dell'applicazione un compilatore la traduzione dello shader del programma in codice eseguibile per la piattaforma di destinazione. **Vulkan fornisce un intermediario binario chiamato SPIR-V (Standard Portable Intermediate Representation), permette la precompilazione degli shader, permette agli sviluppatori di applicazioni di scrivere shader in linguaggi diversi da GLSL, riducendo l'onere sui fornitori di driver.**

- API multipiattaforma supportate sia sui dispositivi mobili sia su schede grafiche di fascia alta.
- OS agnostic per migliorare la portabilità delle applicazioni create utilizzando l'API.
- **Migliorato il supporto per i sistemi moderni che utilizzano multithreading.**
- Ridotto il carico sulla CPU in situazioni in cui la CPU costituisce il collo di bottiglia, permettendo un throughput più elevato per i calcoli GPU e rendering.

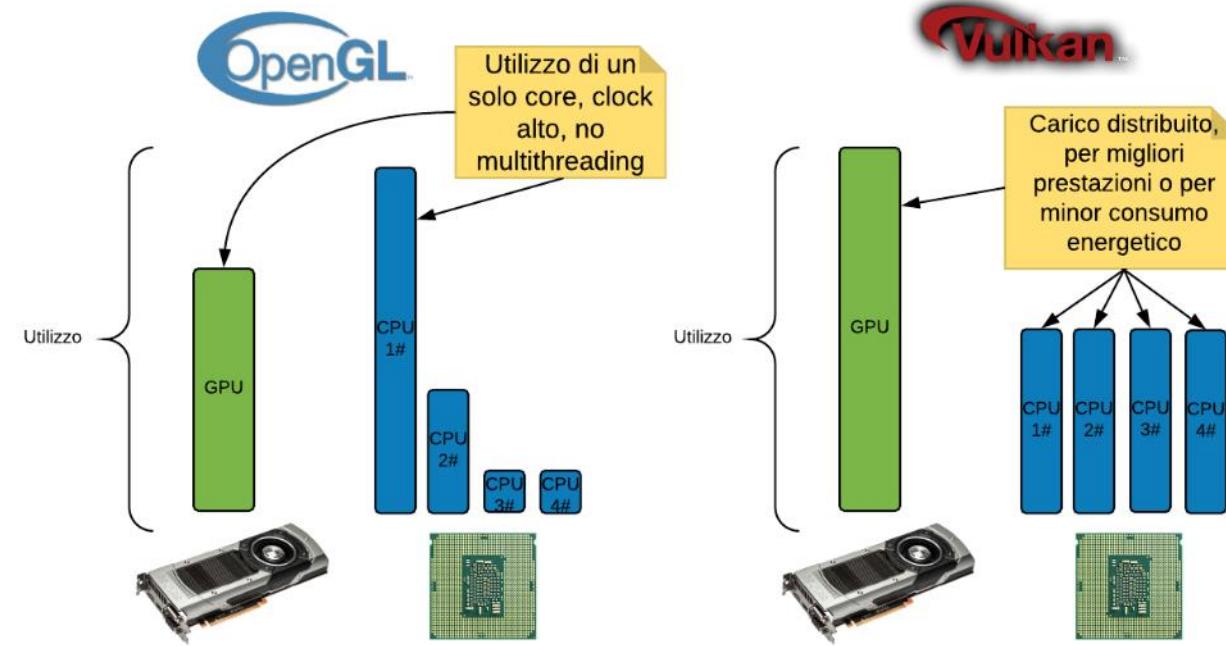


Figura 2.2: Vulkan permette di massimizzare l'utilizzo della GPU grazie al lavoro congiunto di tutti i thread CPU. Inoltre aiuta a minimizzare il clock cpu massimo, contenendo il consumo energetico.



KHRONOS™  
GROUP

# Vulkan Explicit GPU Control

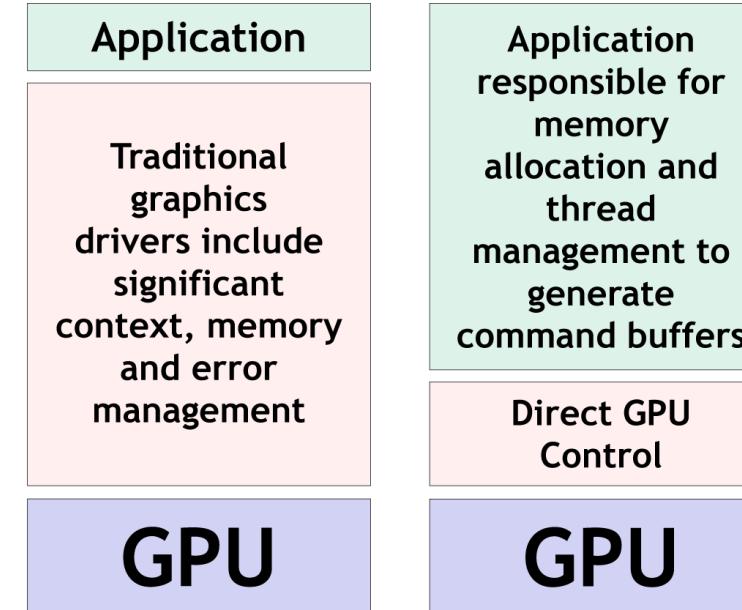


Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets



Vulkan delivers the maximized performance and cross platform portability needed by sophisticated engines, middleware and apps



Progettata per vecchie workstation con rendering diretto.	Progettata per hardware d'avanguardia, comprese le piattaforme cellulari.
Singolo contesto, singola macchina a stati.	Nessun contesto globale, strutturata ad oggetti.
Lo stato è globale e legato al contesto.	Gli stati sono localizzati nei buffer di comando.
Solo operazioni in sequenza.	Operazioni parallele sono possibili.
Gestione della memoria e sincronizzazione nascosta.	Esplicito controllo sulle allocazioni e sulla sincronizzazione.
Controllo degli errori esteso e sempre attivo.	Nessun Controllo, ma c'è la possibilità di agganciare livelli di validazione.
Compilatore degli shader incorporato nel driver.	Compilatore SPIR-V esterno, garantisce maggiore flessibilità ed affidabilità.
Driver complesso, consistenza cross-platform difficile da garantire.	Driver semplice, alta predicitività del codice applicazione



## OpenGL ES

---

- Una versione slim di OpenGL desktop per piattaforme mobili e tablet
- OpenGL ES v 1.x e superiori
- Non permettono la modalità di rendering immediato (glBegin / glEnd)

Modalità di rendering mantenuto mediante array vertex.

- Supporta Texture solo 2D
- Supporta Solo triangoli (GL\_TRIANGLES)

- OpenGL ES 2.0 v

Aggiunto un Supporto Shader simile a quello di OpenGL 2.0 desktop.

- Rimossa completamente la fixed pipeline (cioè senza il supporto di trasformazione matrici, per i materiali e per l'illuminazione)



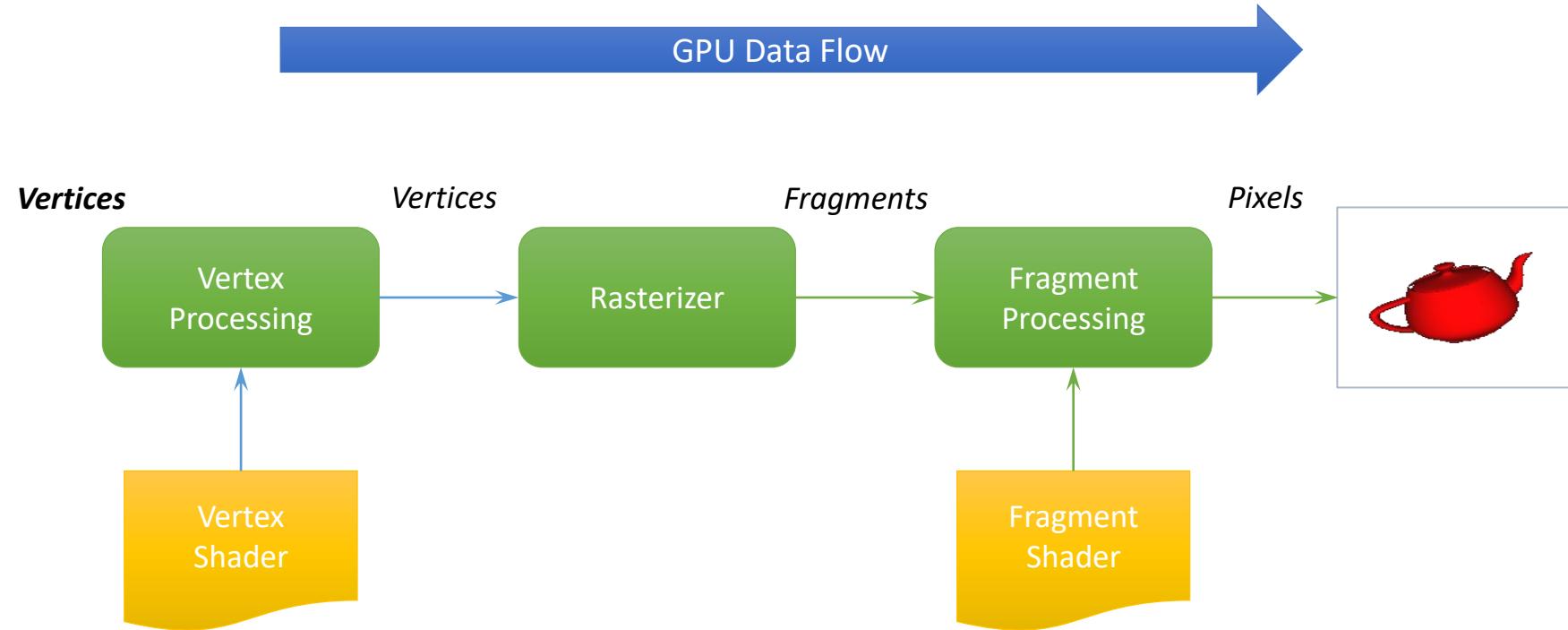
## WebGL

---

- Un sottoinsieme di OpenGL ES che viene eseguito in una finestra del browser senza richiedere alcun plugin
  - Supportato su tutti i principali browser web
  - Accesso diretto al driver OpenGL attraverso l'elemento HTML5 canvas
  - convenzioni di chiamata simili a OpenGL versione desktop
-



# Pipeline di rendering semplificata





---

# Sviluppo di un'applicazione OpenGL

---



## GLUT e FreeGLUT

---

✓ Le applicazioni OpenGL hanno bisogno di **una finestra su schermo su cui visualizzare l'output grafico.**

✓ Hanno bisogno di comunicare con il sistema a finestre nativo.

Ogni sistema a finestre è differente

✓ Usiamo GLUT (più specificamente, freeglut), semplice, libreria open-source che funziona sempre e gestisce tutte le operazioni a finestre: apre finestre, gestisce l'input.

✓ GLUT è stato ideato da Mark Kilgard con l'obiettivo di fornire un kit di strumenti semplici, ma abbastanza potente, per affrontare le complessità del sistema a finestre per la costruzione di applicazioni OpenGL.

✓ GLUT, tuttavia, non è esente da critiche, relative soprattutto alla mancanza di controllo sul ciclo di eventi, ed inoltre GLUT non è stata più aggiornata dalla fine degli anni '90.

✓ Per esempio GLUT non consente l'impostazione contesto, che ci permette di sfruttare la versione di OpenGL compatibile con la scheda video in uso non contempla la rotellina del



GLUT non è open source quindi non potrebbe essere modificata, così sono apparsi cloni GLUT.

---

Questi hanno mantenuto l'API (tutte le funzioni Gluts sono generalmente implementate esattamente con lo stesso nome), ma migliorato ed esteso .

Due delle sostituzioni GLUT più comuni sono **OpenGLUT e Freeglut**. Si tratta di due progetti open source. Freeglut sembra più maturo e più aggiornato, almeno in base alle date di SourceForge per gli ultimi aggiornamenti. In questo corso useremo freeglut.

›Altra possibilità:

›**GLFW**: libreria Open Source, multi-piattaforma per OpenGL, OpenGL ES e sviluppo Vulkan sul desktop. Fornisce una semplice API per creare finestre, contesti e superfici, ricevere input ed eventi.

›GLFW è scritto in C e ha il supporto nativo per Windows, macOS e molti sistemi simili a Unix che usano il sistema X Window, come Linux e FreeBSD.

---



## •OpenGL Extension Wrangler library (GLEW)

---

- GLEW è l'acronimo di "OpenGL Extension Wrangler Library". Si tratta di una libreria open-source che facilita lo sviluppo di applicazioni basate su OpenGL (Open Graphics Library) gestendo le estensioni di OpenGL disponibili sulla piattaforma su cui viene eseguito il software.
- OpenGL è una specifica per la grafica 3D che fornisce un'interfaccia per la programmazione di applicazioni grafiche, ma le implementazioni di OpenGL possono variare tra diversi sistemi operativi e hardware. GLEW viene utilizzato per semplificare la gestione delle estensioni di OpenGL, consentendo agli sviluppatori di verificare quali estensioni sono supportate sulla loro piattaforma e di accedere alle nuove funzionalità di OpenGL in modo uniforme.
- In pratica, GLEW fornisce una serie di funzioni per caricare dinamicamente le estensioni OpenGL supportate, consentendo agli sviluppatori di scrivere codice che funziona su diverse piattaforme senza dover scrivere codice specifico per ciascuna di esse. È una libreria molto utile per gli sviluppatori di giochi e applicazioni grafiche che vogliono assicurarsi che il loro software possa



# OpenGL #include

---

- OpenGL    `#include <GL/gl.h>`
  - Il core della libreria che è platform independent
- GLU        `#include <GL/glu.h>`
  - Una libreria ausiliaria che permette di manipolare una grande varietà di funzioni grafiche accessorie.
- GLUT      `#include <GL/freeglut.h>`
  - una libreria ausiliaria che gestisce la creazione di finestre, chiamate di sistema (pulsanti del mouse, movimento, tastiera, ecc.), callback–
- GLEW      `#include <GL/glew.h>`



# Come inserire le librerie OpenGL nel proprio progetto Visual Studio 2022 :

Visual Studio 2022

Apri recenti

|

◀ Settimana corrente

- ProgettoCostantini.sln 21/09/2023 09:30  
E:\Progetti3d\_22\_23\Progetto3DCostantini\ProgettoCostantini
- Mesh\_Texture.sln 21/09/2023 09:21  
C:\Users\damiana.lazzaro\Desktop\Mesh\_Texture
- doodino.sln 21/09/2023 08:06  
E:\...\ConsegneCG\_Novembre\_21\ConsegneCG\Galavotti\doodleJumpGL-windows

◀ Mese corrente

- CG.sln 18/09/2023 16:53  
E:\...\ComputerGraphicsAssignments-main
- ConsoleApplication1.sln 18/09/2023 16:52  
E:\...\ComputerGraphicsAssignments-Anny\_Bevilacqua\ConsoleApplication1
- Downloads 12/09/2023 17:31  
C:\Users\damiana.lazzaro

Inizia subito

**Clona un repository**  
Consente di ottenere il codice da un repository online, come GitHub o Azure DevOps

**Apri un progetto o una soluzione**  
Consente di aprire un progetto locale o un file con estensione sln di Visual Studio

**Apri una cartella locale**  
Consente di esplorare e modificare il codice in qualsiasi cartella

**Crea un nuovo progetto**  
Consente di scegliere un progetto modello con scaffolding del codice per iniziare

[Continua senza codice →](#)



## Crea un nuovo progetto

Modelli di progetto recenti

Qui verrà visualizzato un elenco dei modelli usati di recente.

App console

Consente di eseguire il codice in un terminale Windows. Stampa "Hello World" per impostazione predefinita.

C++ Windows Console

Progetto CMake

Consente di creare app C++ moderne e multipiattaforma che non dipendono da file con estensione sln o vcxproj.

C++ Windows Linux Console

Creazione guidata applicazione desktop di Windows

Consente di creare l'app di Windows con una procedura guidata.

C++ Windows Desktop Console Libreria

Applicazione desktop di Windows

Progetto per un'applicazione con interfaccia utente grafica eseguita in Windows.

C++ Windows Desktop

Applicazione Python

Progetto per la creazione di un'applicazione da riga di comando

[Indietro](#) [Avanti](#)



Configura il nuovo progetto

App console C++ Windows Console

Nome del progetto

Percorso ...

Nome soluzione

Inserisci soluzione e progetto nella stessa directory

Progetto verrà creato in "C:\Users\damiana.lazzaro\Desktop\CG\_23\_24\CG\_Progetto1\CG\_Progetto1\"

Indietro Crea



File Modifica Visualizza GIT Progetto Compilazione Debug Test Analizza Strumenti Esterioni Finestra ? | 🔎 Cerca ▾ CG\_Progetto1

Accedi 🔒 Live Share 🔍

Tabulazioni CG\_Progetto1 CG\_Progetto1 CG\_Progetto1.cpp

```
CG_Progetto1.cpp : Questo file contiene la funzione 'main', in cui inizia e termina l'esecuzione del programma.

1 // CG_Progetto1.cpp : Questo file contiene la funzione 'main', in cui inizia e termina l'esecuzione del programma.
2 //
3 //
4 #include <iostream>
5
6 int main()
7 {
8     std::cout << "Hello World!\n";
9 }
10
11 // Per eseguire il programma: CTRL+F5 oppure Debug > Avvia senza eseguire debug
12 // Per eseguire il debug del programma: F5 oppure Debug > Avvia debug
13
14 // Suggerimenti per iniziare:
15 // 1. Usare la finestra Esplora soluzioni per aggiungere/gestire i file
16 // 2. Usare la finestra Team Explorer per connettersi al controllo del codice sorgente
17 // 3. Usare la finestra di output per visualizzare l'output di compilazione e altri messaggi
18 // 4. Usare la finestra Elenco errori per visualizzare gli errori
19 // 5. Passare a Progetto > Aggiungi nuovo elemento per creare nuovi file di codice oppure a Progetto > Aggiungi elemento esistente per aggiungere file di codice esistenti al progetto
20 // 6. Per aprire un nuovo questo progetto in futuro, passare a File > Apri > Progetto e selezionare il file con estensione sln
21
```

Non sono stati trovati problemi

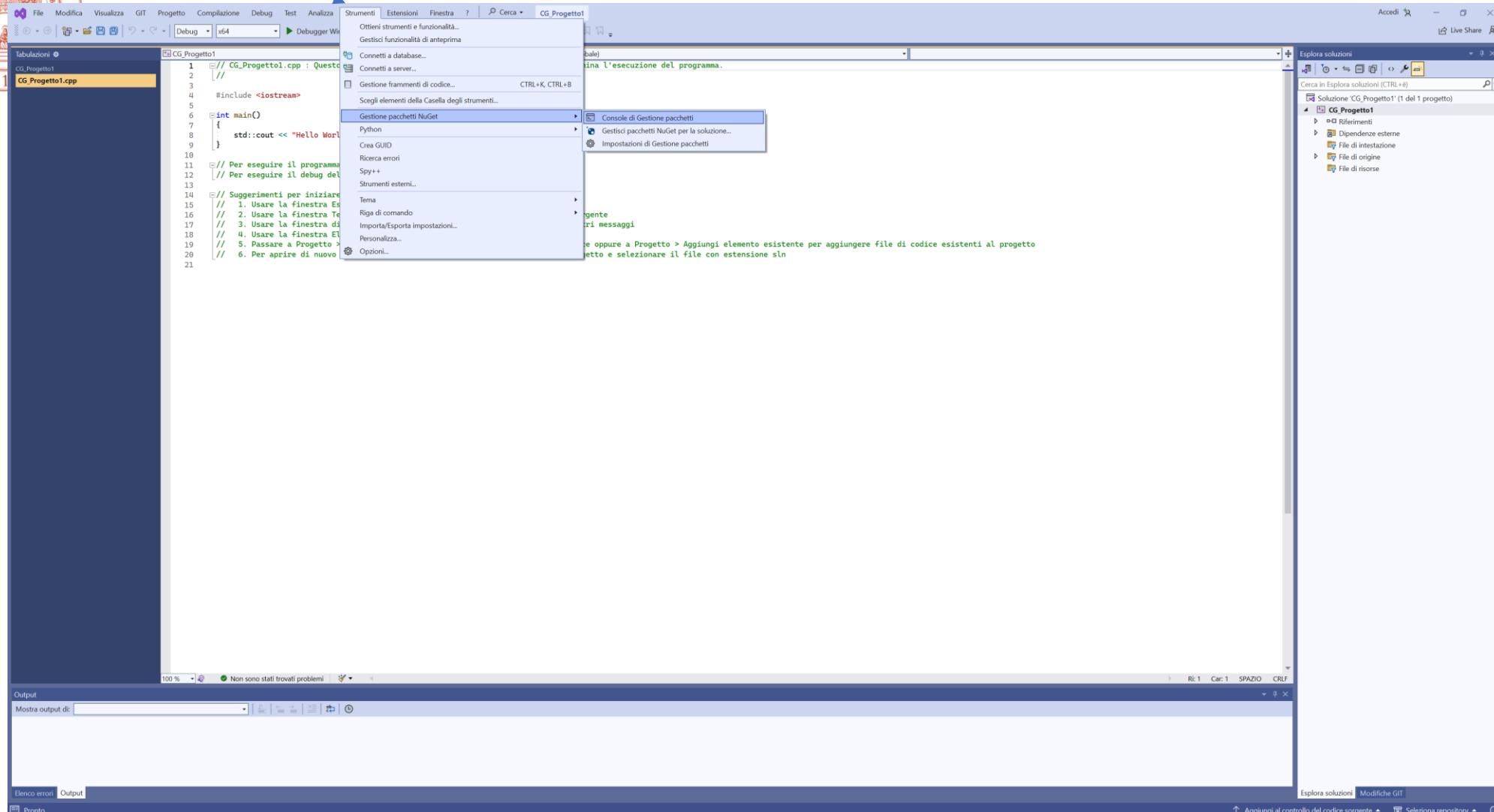
Output

Mostra output di: Elenco errori Output

Creazione del progetto 'CG\_Progetto1' in corso... creazione del progetto completata.

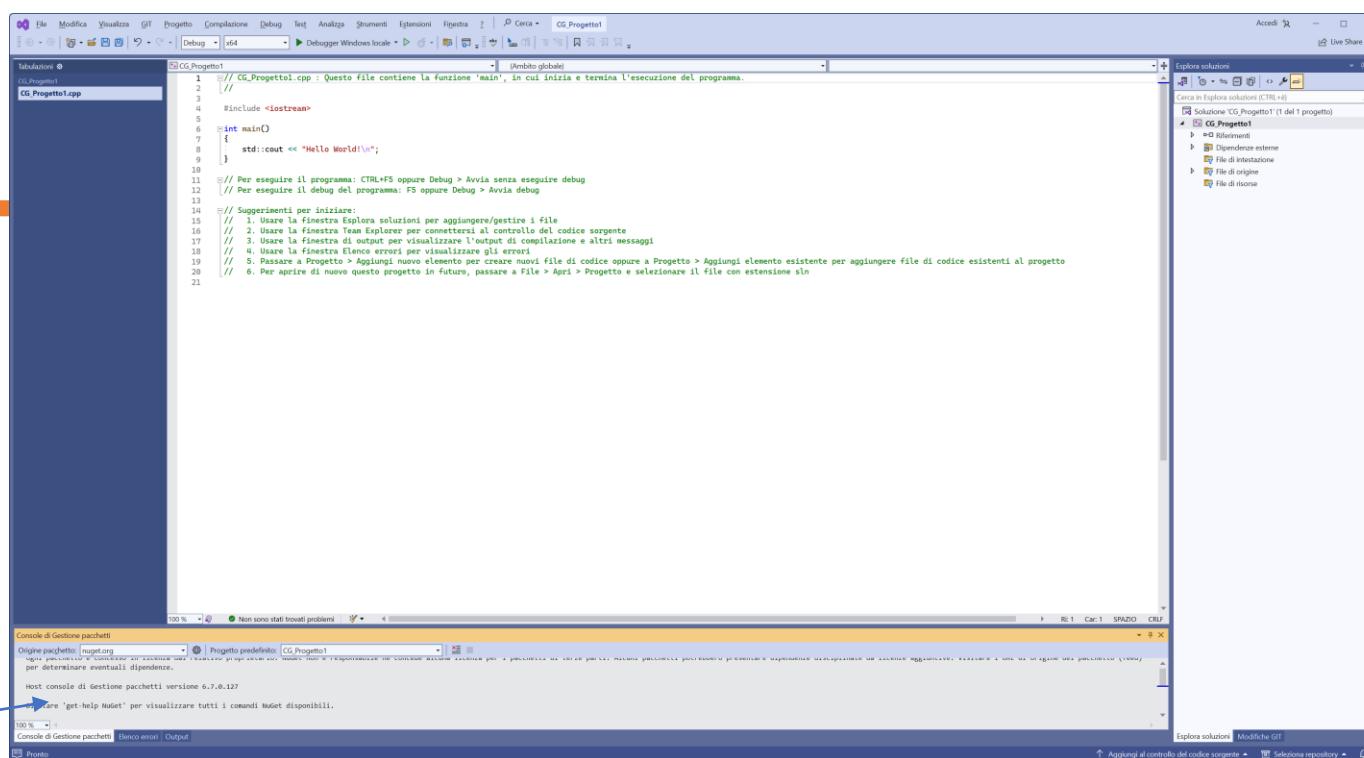
Aggiungi al controllo del codice sorgente Selezione repository

Esplora soluzioni Modifiche GIT



andare in

✓Strumenti-> Gestione pacchetti Nuget->, Console di Gestione pacchetti ->



nel prompt dei comandi digitare

## Install-Package nupengl.core

questo comando crea nel progetto attivo la cartella

\packages\nupengl.core.0.1.0.1\build\native

dove sono contenute le cartella include e lib con i file .h e .lib delle librerie OpenGL e freeglut che vogliamo usare.



## › Programmazione window-based

---

- ✓ I programmi window-based sono “**controllati dagli eventi**”.
  - ✓ Il programma risponde a vari eventi, come il click del mouse, la pressione di un tasto sulla tastiera, il ridimensionamento della finestra.
  - ✓ Il sistema gestisce automaticamente **una coda di eventi**, che riceve messaggi che affermano che sono avvenuti certi eventi.
  - ✓ Risponde agli eventi secondo il criterio “**primo avvenuto, prima servito**”.
  - ✓ **Callback functions:** funzioni di risposta che vengono eseguite quando avvengono degli eventi
  - ✓ Viene creata una funzione di risposta per ogni tipo di evento che può avvenire.
  - ✓ Quando il sistema rimuove un evento dalla coda, esegue semplicemente la funzione di risposta associata con quel tipo di evento.
-



## Scheletro di funzione main per un programma controllato dagli eventi.

```
~#include <cmath>
~#include <iostream>
~#include <GL/glew.h>
~#include <GL/freeglut.h>      (---- include automaticamente gl.h)

~ void main(int argc, char* argv[])
{
    inizializza il toolkit
    Inizializza il contesto
    crea una finestra sullo schermo
    glutDisplay Func(myDisplay);
    }
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutMainLoop();
}
```



#### ◦**glutDisplayFunc(myDisplay)**

Registra la funzione myDisplay come callback function per un evento di ridisegno.

#### ◦**glutReshapefunc( myReshape)**

La finestra dello schermo può essere ridimensionata dall’utente, trascinando con il mouse un corner della finestra in una nuova posizione. La funzione myReshape viene registrata come callback function per un evento di reshape.

#### ◦**glutMousefunc(myMouse)**

Quando uno dei bottoni viene premuto o rilasciato, viene determinato un evento del mouse. Alla funzione myMouse vengono passati gli argomenti che descrivono la locazione del mouse e la natura dell’azione iniziata premendo il bottone.

#### ◦**glutKeyboardFunc(myKeyboard)**

Questo comando registra la funzione myKeyboard( ) con l’evento di premere o lasciare qualche tasto della tastiera.



---

Se un particolare programma non usa il mouse, la corrispondente funzione di risposta non deve essere scritta. Allora il click del mouse non ha effetto sul programma. Lo stesso è vero per i programmi che non usano la tastiera.

`\ glutMainLoop( ).`

Quando questa istruzione viene eseguita, il programma disegna l'immagine iniziale ed entra in un loop senza fine, in cui aspetta semplicemente che avvenga un evento.

---



- 
- ✓ La struttura che devono seguire i programmatori nel gestire gli eventi è “non fare nulla finchè non accade un evento, quando l’evento accade fare l’azione specificata”
  - ✓ La modalità con cui associare una funzione di risposta ad un particolare tipo di evento è spesso dipendente dal sistema.
  - ✓ Useremo la libreria FreeGlut che fornisce strumenti che permettono la gestione degli eventi.

### Creare una finestra

- ✓ La prima cosa da fare prima di iniziare a produrre grafica è creare un contesto OpenGL e una finestra dell'applicazione su cui disegnare. Tuttavia, tali operazioni sono specifiche per sistema operativo e OpenGL cerca intenzionalmente di astrarre se stesso da queste operazioni.



# Core-profile vs modalità immediata

---

- Inizialmente usare OpenGL significava creare applicazioni in modalità **immediate** (spesso indicato come pipeline fixed) che era un metodo facile da usare per disegnare grafica. La maggior parte delle funzionalità di OpenGL era nascosta all'interno della libreria e gli sviluppatori non avevano molto controllo su come OpenGL esegue i suoi calcoli.
- La modalità immediata è davvero facile da usare e da capire, ma è anche estremamente inefficiente. Per questo motivo la specifica OpenGL ha iniziato a **deprecare** la funzionalità in modalità immediata dalla versione 3.2 in poi e ha iniziato a motivare gli sviluppatori a sviluppare in OpenGL **in modalità core-profile**, che ha rimosso tutte le funzionalità obsolete.
- Quando si utilizza il **core-profile**, OpenGL genera un errore e interrompe il disegno quando vengono utilizzate funzionalità deprecate di OpenGL.



# Creare contesti OpenGL con i profili

. Ciò si ottiene con le seguenti funzioni di freeglut:

*glutInitContextVersion (int MajorVersion, int MinorVersion)*

*glutInitContextProfile (int profile)*

*glutInitContextFlags (GLuint flags)*

I valori possibili per il *profile* sono:

**GLUT\_CORE\_PROFILE**

e

**GLUT\_COMPATIBILITY\_PROFILE.**

I *flag* possono essere impostati a

**GLUT\_DEBUG e / o GLUT\_FORWARD\_COMPATIBLE.**

- ✓ GLUT\_CORE\_PROFILE non include alcuna delle caratteristiche che sono state rimosse nelle versioni precedenti,
- ✓ GLUT\_COMPATIBILITY\_PROFILE le include.
- ✓ Una implementazione OpenGL garantisce sempre di contenere il Core Profile, ma non sempre un Compatibility Profile (il profilo che è compatibile con le più vecchie funzionalità OpenGL.)



- Se si imposta il flag **GLUT\_FORWARD\_COMPATIBLE**, il contesto che viene restituito **non conterrà nessuna delle caratteristiche che sono state deprecate nella versione richiesta**, rendendo così compatibile con le versioni future che possono aver rimosso queste caratteristiche.
- Se si imposta il flag **GLUT\_DEBUG**, verrà restituito un contesto di debug che includerà controlli aggiuntivi, la validazione, e altre funzionalità che possono essere utili durante lo sviluppo
- I flag sopra possono essere combinati insieme, con eccezione del Core **GLUT\_CORE\_PROFILE** e **GLUT\_COMPATIBILITY\_PROFILE**
- La combinazione di flag viene fatta usando l'operatore OR bitwise, il simbolo pipe (|) in C.
- Ad esempio per impostare un contesto di **CORE\_PROFILE** con OpenGL 4.2 basta scrivere:

```
glutInitContextVersion(4, 2);  
glutInitContextProfile(GLUT_CORE_PROFILE);
```



```
void main(int argc, char* argv[])
{
    glutInit(&argc, argv);

    glutInitContextVersion(4, 0);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(800, 800);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Prima Applicazione OpenGL");
    glutDisplayFunc(drawScene);

    glewExperimental = GL_TRUE; glewExperimental è una variabile già definita da GLEW. Deve essere impostata a GL_TRUE prima di chiamare glewInit().
    glewInit(); //Inizializza l'estensione utilizzata

    glutMainLoop();
}
```



## Inizializzazione e visualizzazione di una finestra sullo schermo.

---

*glutInit(&argc,argv);* inizializza il FreeGLUT toolkit

*glutInitContextVersion(4, 0); Inizializza la versione del Contesto  
glutInitContextProfile(GLUT\_CORE\_PROFILE); Inizializza il Profile*

*glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);*

Questa funzione specifica come deve essere inizializzato il display. Le costanti GLUT\_SINGLE e GLUT\_RGB, di cui è fatto l' “or” ( | ), indicano che deve essere allocato un Buffer Display unico e che i colori saranno specificati utilizzando quantità desiderate di rosso, di verde e di blu.

✓*glutInitWindowSize(800,800);*

Questa funzione specifica che inizialmente la finestra dello schermo sarà larga 640 pixels e alta 480 pixels. L'utente può ridimensionare la finestra come desidera, quando il programma gira.

---



*-glutInitWindowPosition(100,100);*

Questa funzione specifica che il corner più in alto a sinistra della finestra sarà posizionato sullo schermo 100 pixel in su rispetto dal bordo sinistro e 100 pixel in giù dal top.

*-glutCreateWindow("Prima applicazione OpenGL");*

Questa chiamata apre e visualizza la finestra sullo schermo, dandogli il titolo specificato tra “ ”.



## Introduzione agli shader

---

- ✓ Sono speciali micro-programmi che vengono eseguiti sulla GPU.
- ✓ Consentono la programmabilità delle diverse fasi della moderna pipeline grafica GPU
- ✓ OpenGL (versione inferiore a 3.0) supporta solo due tipi di shader: vertex e fragment shader
- ✓ OpenGL moderna (3.0 e versioni successive) supporta vertex, geometry e fragment shader.
- ✓ La corrente versione di OpenGL (v 4.0 e versioni successive) supporta vertex, tessellation control/evaluation shaders, geometry and fragment shaders

Gli shader convenzionalmente sono stati utilizzati per lavori di grafica solo con processori programmabili dedicati (processori vertex e fragment)

---



- Sempre più persone hanno utilizzato gli shader per lavori non di grafica (GPGPU) per esempio l'elaborazione delle immagini, il calcolo di AI, dinamica (2004-2006).
- A causa di questa tendenza, NVIDIA ha proposto un framework GPGPU basato su C chiamato CUDA nel 2007. E' stato introdotto un modello di stream processor, che è un processore general purpose.
- Il gruppo Khronos ha poi creato una specifica aperta OpenCL nel 2008.

Le GPU moderne possono funzionare in due modalità: modalità -compute o modalità -shader

- **Modalità Compute**

-I Processori della GPU funzionano come processori general purpose che li rende adatti per GPGPU

- **Modalità Shader**

-I processori della GPU lavorano come processori specializzati per elaborazione di vertici, tessellation, geometria, frammenti che li rende adatti per elaborazione grafica

• La GPU può essere solo in un'modalità, compute o shader, non può essere utilizzata in modalità differenti nello stesso istante



- Ad oggi, la versione disponibile di OpenGL è la 4.6, Tutte le versioni di OpenGL a partire dalla 3.3 aggiungono funzionalità extra utili a OpenGL senza cambiare la meccanica di base di OpenGL; le versioni più recenti introducono solo modi leggermente più efficienti o più utili per svolgere le stesse attività.
- Quando si utilizzano le funzionalità della versione più recente di OpenGL, solo le più moderne schede grafiche saranno in grado di eseguire l'applicazione. Questo è spesso il motivo per cui la maggior parte degli sviluppatori di solito prende come target versioni inferiori di OpenGL e abilita facoltativamente funzionalità delle versioni superiori.
- **Estensioni**
- Una grande caratteristica di OpenGL è il supporto delle estensioni. Ogni volta che un'azienda grafica presenta una nuova tecnica o una nuova grande ottimizzazione per il rendering, questa si trova spesso in un' estensione implementata nei driver.



---

Se l'hardware su cui viene eseguita un'applicazione supporta tale estensione, lo sviluppatore può utilizzare la funzionalità fornita dall'estensione per una grafica più avanzata o efficiente. In questo modo, uno sviluppatore grafico può ancora utilizzare queste nuove tecniche di rendering, senza dover aspettare che OpenGL includa la funzionalità nelle sue versioni future, semplicemente controllando se l'estensione è supportata dalla scheda grafica. Spesso, quando un'estensione è popolare o molto utile, alla fine diventa parte delle future versioni di OpenGL.

- **Macchina a stati**
  - OpenGL è una grande macchina a stati: una raccolta di variabili che definiscono come OpenGL deve funzionare in ogni momento. Lo stato di OpenGL viene comunemente definito OpenGL context . Quando si utilizza OpenGL, spesso cambiamo il suo stato impostando alcune opzioni, manipolando alcuni buffer e quindi eseguendo il rendering utilizzando il contesto corrente.
-



---

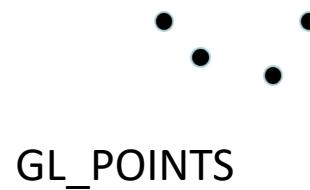
Ogni volta che diciamo a OpenGL che ora vogliamo disegnare linee anziché triangoli, cambiamo lo stato di OpenGL cambiando alcune variabili di context che impostano come OpenGL dovrebbe disegnare. Non appena cambiamo il contesto dicendo a OpenGL che dovrebbe tracciare linee, i comandi di disegno successivi disegneranno linee invece di triangoli.

---



# Primitive geometriche in OpenGL

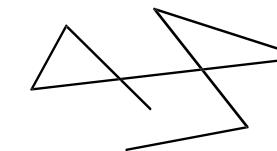
Tutte le primitive sono specificate mediante vertici



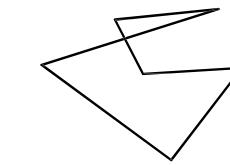
GL\_POINTS



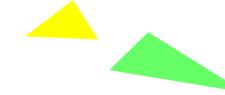
GL\_LINES



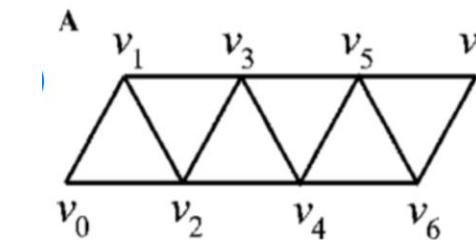
GL\_LINE\_STRIP



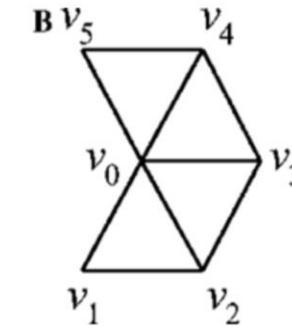
GL\_LINE\_LOOP



GL\_TRIANGLES



GL\_TRIANGLE\_STRIP



GL\_TRIANGLE\_FAN



OpenGL visualizza solo triangoli

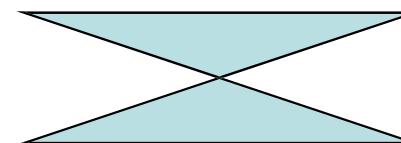
Semplici: i bordi non possono incrociarsi

Convessi: tutti i punti sul segmento di linea tra due punti in un poligono sono anche nel poligono

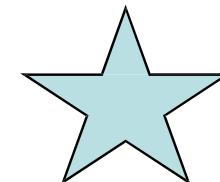
Flat : tutti i vertici sono sullo stesso piano

Il programma applicativo deve tassellare un poligono in triangoli (triangolazione)

OpenGL 4.1 contiene un tassellatore



Poligono non semplice



Poligono non convesso



# Diversi tipi di rendering pipeline

---

## 1) Immediate mode o fixed pipeline

- Ogni volta che un vertice viene specificato dall'applicazione, la sua posizione viene spedita alla GPU.
  - Fa uso di `glVertex`, `glBegin/glEnd`
  - Crea un collo di bottiglia tra CPU e GPU
  - Per disegnare gli stessi dati, bisogna spedirli nuovamente alla GPU.
-



# Immediate Mode

## Formato delle funzioni OpenGL

rtype **glNAME{1234}{b s i f d ub us ui}[v]**

Appartiene alla libraria GL, nome funzione, dimensioni,

void **glVertex3fv( v )**

v è un puntatore ad un array

*Number of components*

2 - (x, y)  
3 - (x, y, z)  
4 - (x, y, z, w)

*Data Type*

b - byte  
ub - unsigned byte  
s - short  
us - unsigned short  
i - int  
ui - unsigned int  
f - float  
d - double

*Vector*

omit "v" for scalar form

**glVertex2f( x, y )**



## Immediate Mode: Costruire una primitiva

```
glBegin(primitiveType)
...
...
glEnd();
```

```
GLfloat red, green, blue;
GLfloat coords[3];

glBegin (primitiveType); % primitiveType Tipo della primitiva geometrica

for ( i = 0; i < nVertices; ++i )
{
    glColor3f( red, green, blue );
    glVertex3fv( coords );
}
glEnd();
```



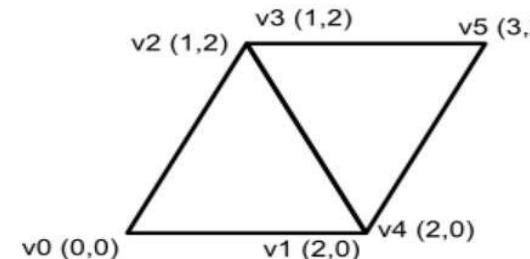
## 2) DrawArrays/DrawElements with VBOs

I Vertex Buffer Object (VBO) memorizzano array di dati di vertici, posizionali o descrittivi. Con un VBO è possibile calcolare tutti i vertici contemporaneamente, comprimerli in un VBO e passarli in massa a OpenGL per consentire alla GPU di elaborare tutti i vertici insieme.

`glDrawArrays( )`

**DrawArrays (senza indexing)**

% riduce il numero di chiamate a funzione

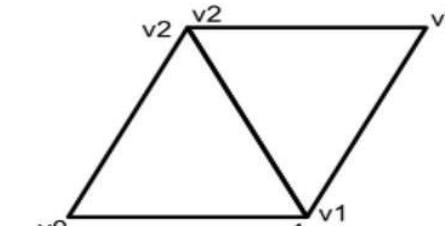


`[0,0, 2,0, 1,2, 1,2, 2,0, 3,2]`

`glDrawElements( )`

**DrawElements (con indexing)**

% riduce il numero di chiamate a funzione e l'uso ridondante di vertici condivisi.



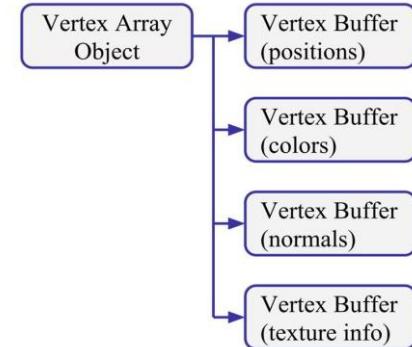
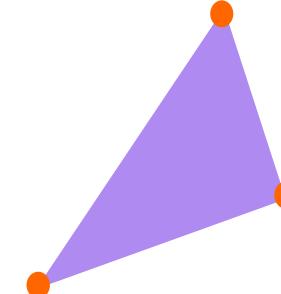
`[0,1,2, 2,1,3]  
[0,0, 2,0, 1,2, 3,2]`



- 
- ✓ Il vantaggio dell'uso dei VBO è che possiamo inviare grandi quantità di dati contemporaneamente alla scheda grafica e tenerli lì se c'è abbastanza memoria, senza dover inviare dati un vertice alla volta.
  - ✓ L'invio di dati alla scheda grafica dalla CPU è relativamente lento.
  - ✓ Una volta che i dati sono nella memoria della scheda grafica, il Vertex Shader ha accesso quasi istantaneo ai vertici in maniera estremamente veloce
-

# Vertex Buffer Object (VBO)

- Gli oggetti geometrici sono rappresentati usando i vertici.
- Un vertice è una raccolta di attributi generici
  - coordinate posizionali
  - colori
  - coordinate di texture
  - qualsiasi altro dato associato a quel punto nello spazio
- la posizione viene memorizzata in coordinate omogenee a 4 dimensioni
- I dati dei vertici devono essere archiviati nei **Vertex Buffer Object (VBO)**
- Uno o più VBO devono essere archiviati in **Vertex Array Object (VAO)**



Possiamo creare tutti gli oggetti buffer VBO che vogliamo per diversi tipi di dati per vertice. Questo ci permette di legare i vertici con le normali, i colori, le coordinate delle texture, ecc...



# VERTEX ARRAY OBJECTS (VAO)

---

I VAO memorizzano i dati di un oggetto geometrico

## Passaggi nell'utilizzo di un VAO

```
unsigned int my_vao;
```

1) generare il nome del VAO chiamando la funzione

```
glGenVertexArrays( 1, &my_vao );
```

2) impostare il VAO come attivo

```
glBindVertexArray(my_vao);
```

3) aggiornare i VBO associati a questo VAO

4) associare (bind) il VAO per usarlo nel rendering

Questo approccio consente a una singola chiamata di funzione di specificare tutti i dati per un oggetto

---



# VBO: memorizzazione degli attributi dei vertici

---

I dati dei vertici devono essere archiviati in un VBO, questo deve essere fatto solo una volta, e quindi associati a un VAO,

Per inizializzare un VBO e caricare il vettore di dati in esso:

- In fase di inizializzazione

– generare i nomi dei VBO mediante la chiamata:

```
unsigned int VBO_1;
```

```
glGenBuffers(1, &VBO_1);
```

– settare un buffer come current buffer, cioè buffer su cui stiamo attualmente lavorando

```
glBindBuffer(GL_ARRAY_BUFFER, VBO_1);
```

– caricare il vettore dei dati (vData) nel VBO (sulla memoria della GPU) usando:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vData), vData, GL_STATIC_DRAW);
```

- In fase di rendering (in drawScene())

– bind VAO per usarlo in fase di rendering

```
glBindVertexArray(vao)
```

```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```





---

Un primo programma:  
Disegnare un triangolo

---



# Programmazione OpenGL in breve:

---

- Per programmare applicazioni grafiche in OpenGL bisogna essenzialmente seguire i seguenti passi
    - Creare shader programs
    - Creare buffer objects e caricare i dati dentro di essi
    - Collegare le posizioni dei dati con le variabili degli shaders
    - Renderizzare
-



- 
- In OpenGL tutto è nello spazio 3D, ma lo schermo o la finestra è una matrice 2D di pixel, quindi gran parte del lavoro di OpenGL riguarda la trasformazione di tutte le coordinate 3D in pixel 2D che si adattano allo schermo. Il processo di trasformazione delle coordinate 3D in pixel 2D è gestito dalla pipeline grafica di OpenGL.

La **pipeline grafica** può essere divisa in due grandi parti:

- la prima trasforma le coordinate 3D in coordinate 2D (sottosistema geometrico)
- la seconda parte trasforma le coordinate 2D in pixel colorati reali (sottosistema raster)

**La pipeline grafica prende come input un insieme di coordinate 3D e le trasforma in pixel 2D colorati sullo schermo.**

---



---

La pipeline grafica può essere suddivisa in più passaggi in cui ogni passaggio richiede l'output del passaggio precedente come input.

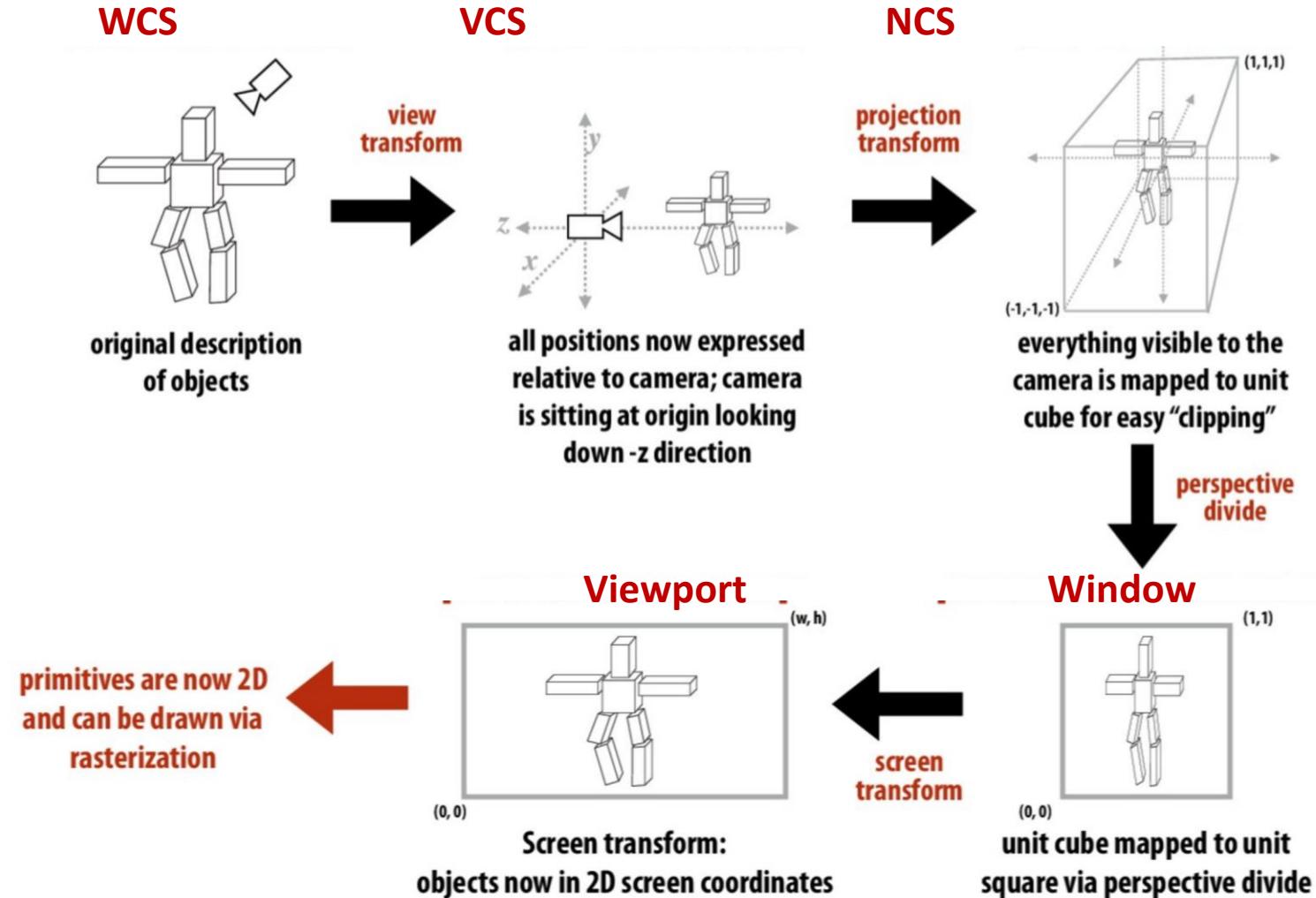
Tutti questi passaggi sono altamente specializzati (hanno una funzione specifica) e possono essere facilmente eseguiti in parallelo. Grazie alla loro natura parallela, le schede grafiche di oggi hanno migliaia di piccoli core di elaborazione per elaborare rapidamente i dati all'interno della pipeline grafica. I core di elaborazione eseguono piccoli programmi sulla GPU per ogni fase della pipeline. Questi piccoli programmi vengono chiamati shaders

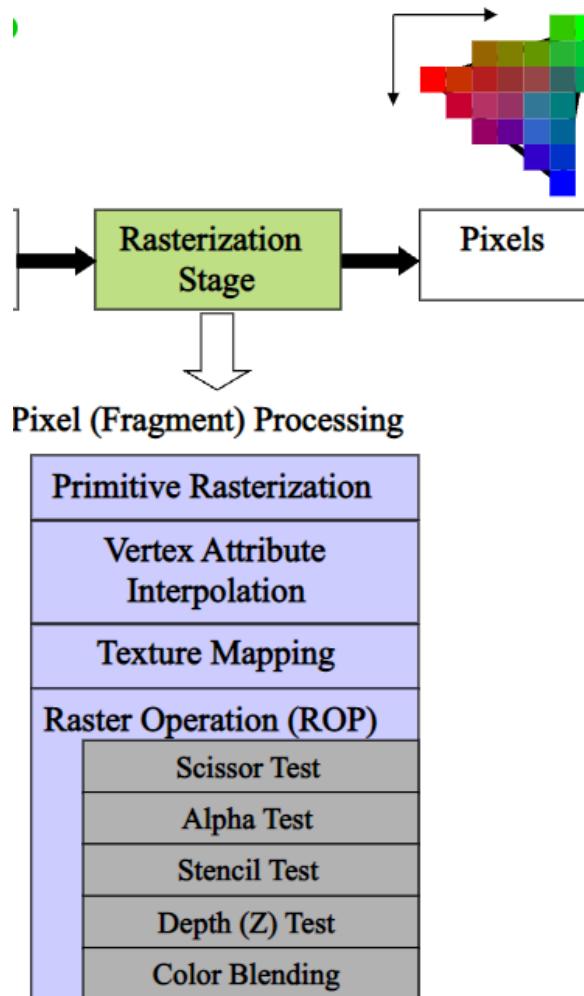
Alcuni di questi shader sono configurabili dallo sviluppatore che può scrivere i propri shader per sostituire gli shader predefiniti esistenti. Questo ci dà un controllo molto più preciso su parti specifiche della pipeline e poiché funzionano sulla GPU, possono anche farci risparmiare tempo prezioso della CPU. Gli shader sono scritti nel Linguaggio di Shading OpenGL ( GLSL ).

---



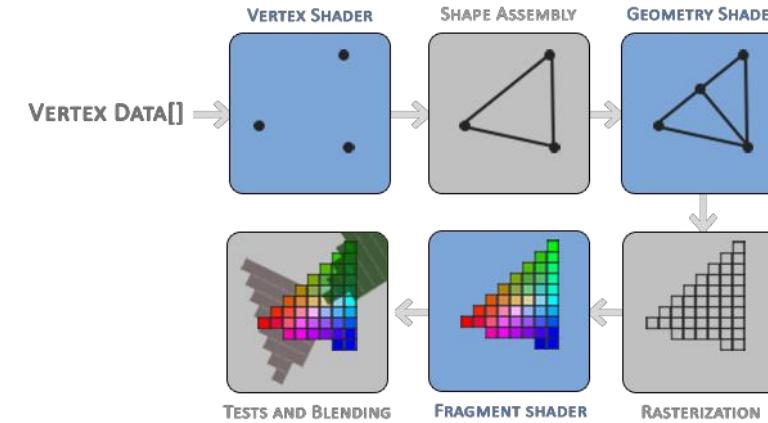
# Geometry stage







Supponiamo di voler disegnare un triangolo, a partire dai suoi tre vertici.



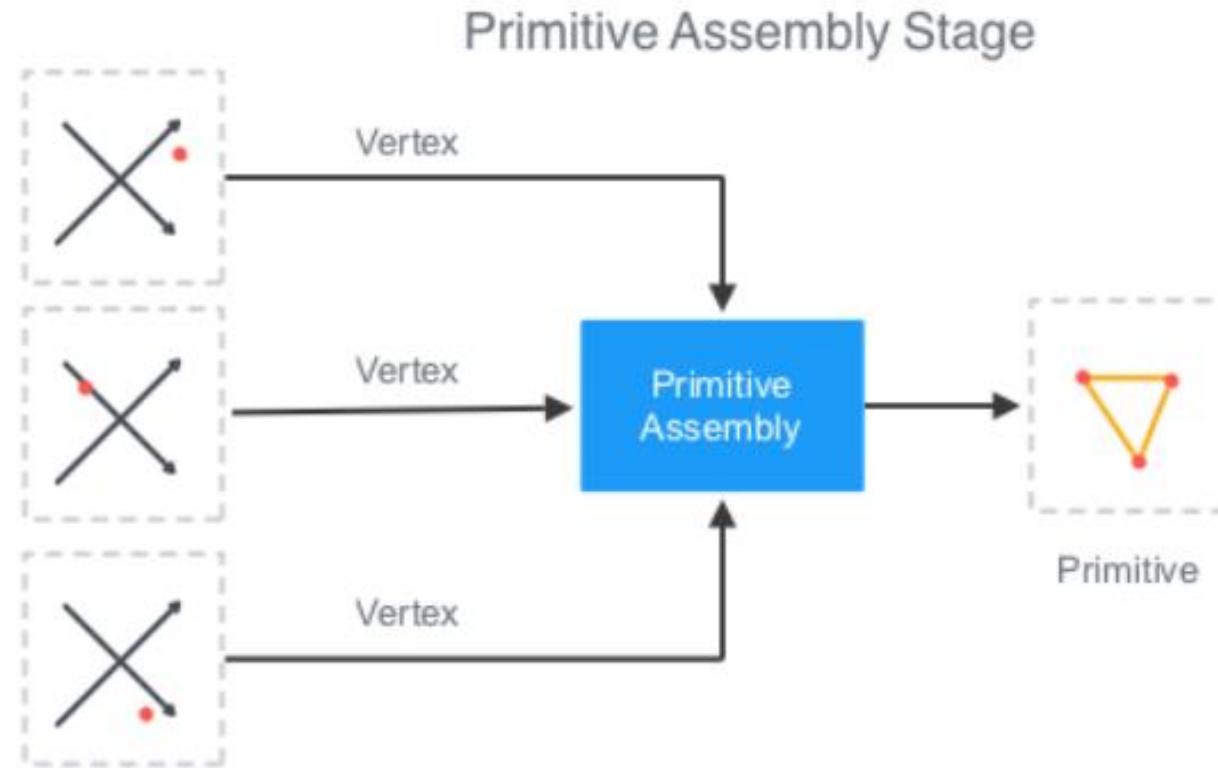
- Come input per la pipeline grafica, passiamo in un elenco di tre coordinate 3D dei vertici di un triangolo in un array chiamato Vertex Data.
- I dati di ogni vertice sono rappresentati usando attributi , per semplicità supponiamo che ogni vertice sia costituito solo da una posizione 3D e da un valore di colore.



- 
- ✓ La prima parte della pipeline è il **Vertex Shader** che accetta **come input un singolo vertice**. Lo scopo principale del Vertex Shader è quello **di trasformare le coordinate 3D in “altre” coordinate 3D** (coordinate di clipping) e il Vertex Shader consente di eseguire alcune elaborazioni di base sugli attributi del vertice.
  - ✓ La fase di **assemblaggio in primitive** accetta come input **tutti i vertici (o il vertice, se si sceglie GL\_POINTS ) dal Vertex Shader** che formano una primitiva ed assembla tutti i punti nella forma primitiva data; in questo caso un triangolo.
-



# Primitive Assembly



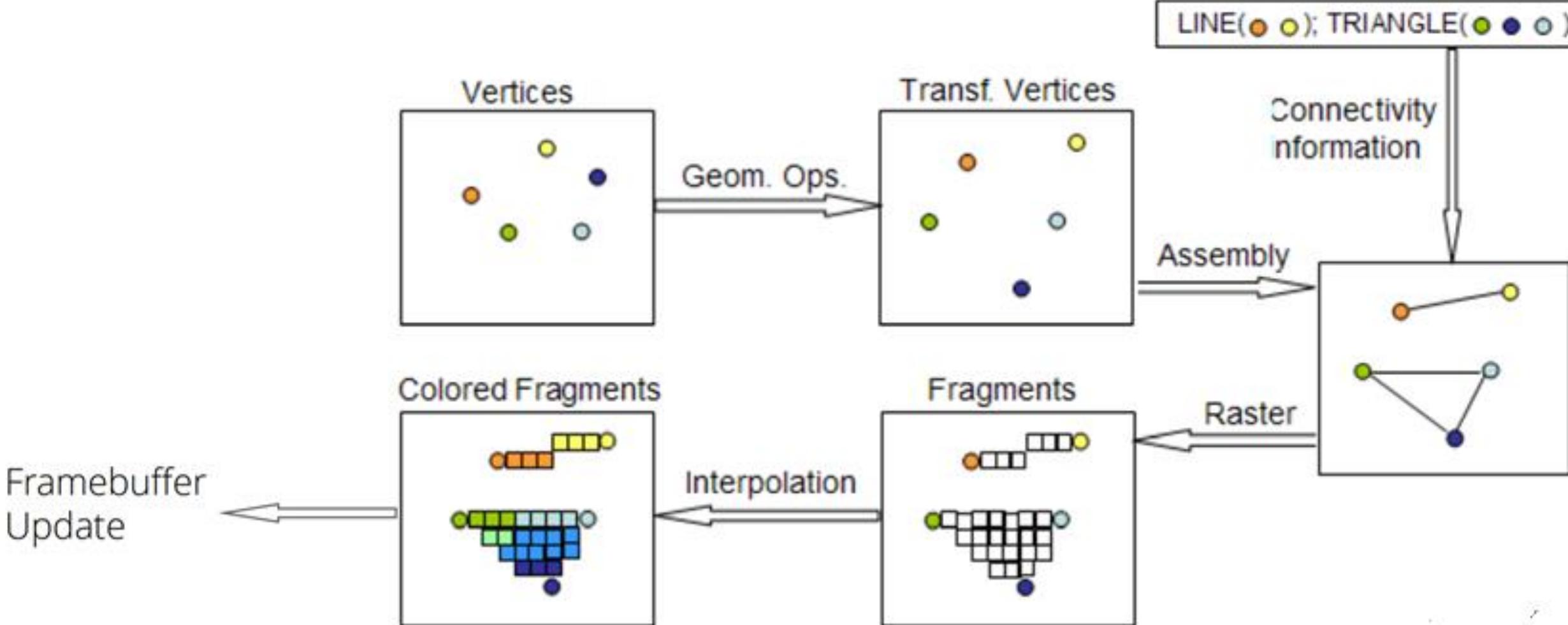
Dopo che tre vertici sono stati elaborati dal vertex shader, vengono portati nella fase di assemblaggio delle primitive. È qui che viene costruita una primitiva collegando i vertici in un ordine specificato.



- ✓ L'output della fase di assemblaggio in primitive viene passato al **Geometry Shader**. Il Geometry Shader prende come input una raccolta di vertici che formano una primitiva e ha **la capacità di generare altre forme emettendo nuovi vertici per formare nuove (o altre) primitive**. Nel caso in figura, genera un secondo triangolo dalla forma data.
- ✓ L'output del Geometry Shader viene quindi passato alla fase di rasterizzazione durante la quale le primitive risultanti vengono mappate sui pixel corrispondenti nella schermata finale, risultando in frammenti, avviene il processo di interpolazione degli attributi sui frammenti a partire dagli attributi sui vertici. Rappresentano l'input per il **Fragment Shader**.
- ✓ Lo scopo principale del **Fragment Shader** è calcolare il colore finale di un pixel e di solito questo è lo stadio in cui si verificano tutti gli effetti OpenGL avanzati.
- ✓ Di solito il Fragment Shader contiene informazioni sulla scena 3D utili per calcolare il colore finale dei pixel (come luci, ombre, colore della luce e così via).



- 
- ✓ Dopo che tutti i valori di colore corrispondenti sono stati determinati, l'oggetto finale passerà attraverso un ulteriore stadio che chiamiamo *alfa-test* e *blending*. Questa fase controlla il corrispondente valore di profondità (e stencil) del frammento e li utilizza per verificare se il frammento risultante si trova davanti o dietro altri oggetti e deve essere scartato di conseguenza.
  
  - ✓ Questa fase controlla anche i valori di alfa (i valori alfa definiscono l'opacità di un oggetto) e miscela gli oggetti di conseguenza.
  - ✓ La pipeline grafica è piuttosto complessa e contiene molte parti configurabili. Tuttavia, per quasi tutti i casi dobbiamo solo lavorare con vertex shader e fragment shader. Il geometry shader è opzionale e di solito viene lasciato allo shader predefinito.
  - ✓ Nelle OpenGL viene richiesto di definire almeno uno shader di vertici e frammenti (non ci sono shader di vertici / frammenti predefiniti sulla GPU).
-





## Vertice

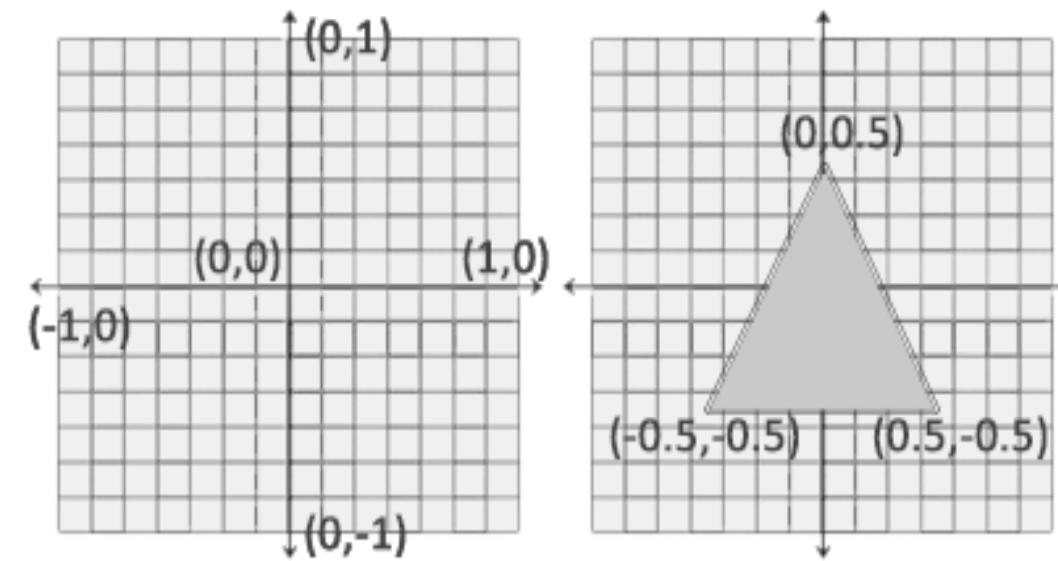
- Specifichiamo 3 vertici ciascuno dei quali individuato da tre coordinate (x,y,z) .
- OpenGL visualizza le coordinate 3D solo quando si trovano in un intervallo specifico tra -1.0 e 1.0 su tutti e 3 gli assi ( x , y z ). Tutte le coordinate all'interno di questo range, detto delle coordinate del dispositivo normalizzate, saranno visibili sullo schermo (quelle al di fuori di questa regione non saranno visibili).
- Nelle applicazioni **reali i dati di input non sono generalmente già nelle coordinate del dispositivo normalizzate**, quindi dobbiamo prima trasformare i dati di input in coordinate che rientrano nella regione visibile di OpenGL.
- Specifichiamo le coordinate dei tre vertici in coordinate del dispositivo normalizzate in un array float

```
Float vertices[] = {-0.5f, -0.5f, 0.0f,  
0.5f, -0.5f, 0.0f, 0.0f, 0.5f, 0.0f};
```

Poiché la terza coordinata è uguale, a zero, per ciascun vertice, allora la profondità del triangolo rimane la stessa e si ha visualizzazione 2D.

## Coordinate dispositivo normalizzate (NDC)

Una volta che le coordinate del vertice sono state elaborate nel Vertex Shader, dovranno essere espresse in coordinate del dispositivo normalizzate. Tutte le coordinate che non rientrano in questo intervallo verranno scartate / troncate e non saranno visibili sullo schermo.





---

Le coordinate NDC verranno quindi trasformate in coordinate dello spazio dello schermo tramite la trasformazione della finestra passando i dati forniti dall'utente alla funzione `glViewport`. Le coordinate dello spazio dello schermo risultanti vengono quindi trasformate in frammenti come input per lo shader di frammenti.

## **Come inviare i dati del vertice come input al primo processo della pipeline grafica: il Vertex Shader?**

Questo viene fatto:

- ✓ creando un VAO sulla GPU in cui archiviamo i dati dei vertici,  
Collegare le posizioni dei dati con le variabili degli shaders

✓

.



Un *Vertex Buffer Object* è un oggetto OpenGL che ha un ID univoco corrispondente a quel buffer.

---

## Generazione di Vertex Buffer Object, con un identificativo, (per esempio VBO):

si utilizza la funzione `glGenBuffers`

```
unsigned int VBO;  
glGenBuffers (1, &VBO);
```

OpenGL ha molti tipi di *Buffer Object* e il tipo di buffer di un *Vertex Buffer Object* è `GL_ARRAY_BUFFER`.

Possiamo associare il buffer appena creato alla destinazione `GL_ARRAY_BUFFER` con la funzione `glBindBuffer`:

```
glBindBuffer (GL_ARRAY_BUFFER, VBO);
```

- Da questo momento in poi, qualsiasi chiamata al buffer che effettuiamo (sulla destinazione `GL_ARRAY_BUFFER`) verrà utilizzata per configurare il buffer attualmente associato, che è `VBO`.
- Quindi possiamo effettuare una chiamata alla funzione `glBufferData` che copia i dati dei vertici precedentemente definiti nella memoria del buffer:



**glBufferData** è una funzione che ha lo scopo di copiare i dati definiti dall'utente nel buffer attualmente associato.

---

```
glBufferData (GL_ARRAY_BUFFER, // tipo di buffer in cui vogliamo copiare i dati  
             sizeof(vertices), // dimensione dei dati (in byte) da passare al buffer  
             vertices, // i dati effettivi che vogliamo inviare  
             GL_STATIC_DRAW // specifica come la scheda grafica deve gestire i dati) ;
```

- Il quarto parametro specifica come vogliamo che la scheda grafica gestisca i dati. Questo può assumere 3 forme:
    - **GL\_STREAM\_DRAW** : i dati vengono impostati una sola volta e utilizzati dalla GPU al massimo alcune volte.
    - **GL\_STATIC\_DRAW** : i dati vengono impostati una sola volta e utilizzati più volte.
    - **GL\_DYNAMIC\_DRAW** : i dati vengono cambiati molto e utilizzati più volte.
-



---

I dati di posizione del triangolo non cambiano, vengono usati molto e rimangono gli stessi per ogni chiamata di rendering, quindi il suo tipo di utilizzo è `GL_STATIC_DRAW`

Se, ad esempio, si avesse un buffer con dati che potrebbero cambiare frequentemente, un tipo di utilizzo di `GL_DYNAMIC_DRAW` garantisce che la scheda grafica inserirà i dati in memoria per consentire scritture più veloci.

- Abbiamo così archiviato i dati dei vertici nella memoria della scheda grafica come gestiti da un Vertex Buffer Object chiamato VBO . Successivamente vogliamo creare un Vertex Shader ed un Fragment Shader che elaborino effettivamente questi dati.

---



## Vertex Shader

- Il Vertex Shader è uno degli shader programmabili.
- OpenGL richiede che almeno impostiamo un Vertex Shader ed un Fragment Shader per poter fare un primo semplice rendering, non c'è un vertex Shader ed un Fragment Shader già predefiniti.
- Introduciamo brevemente gli shader e configuriamo due shader molto semplici per disegnare il nostro primo triangolo.
- La prima cosa che dobbiamo fare è scrivere il Vertex Shader nel linguaggio shader *GLSL* ([OpenGL Shading Language](#)) e quindi compilare questo shader in modo che possiamo utilizzarlo nella nostra applicazione.
- Di seguito il codice sorgente di un semplice Vertex Shader in GLSL:

```
#version 430 core  
layout (location = 0) in vec3 aPos;  
  
void main()  
{  
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);  
}
```



- Ogni shader inizia con una dichiarazione della sua versione.  
Specifichiamo esplicitamente che stiamo utilizzando la funzionalità del core profile.
- Successivamente dichiariamo tutti **gli attributi del vertice di input** nel Vertex Shader con la parola chiave **in**. Al momento ci preoccupiamo solo dei dati di posizione, quindi abbiamo bisogno di un solo attributo vertice.
- GLSL ha un tipo di dati vettoriale che contiene da 1 a 4 float in base alla sua cifra postfissa. Poiché ogni vertice ha una coordinata 3D, creiamo una variabile di input `vec3` con il nome `aPos` .
- Specifichiamo la posizione della variabile di input tramite  
`layout (location = 0)`

## Vettore

Nella programmazione grafica usiamo il concetto matematico di un vettore abbastanza spesso, poiché rappresenta ordinatamente posizioni / direzioni in qualsiasi spazio e ha utili proprietà matematiche. Un vettore in GLSL ha una dimensione massima di 4 e ciascuno dei suoi valori può essere recuperato tramite `vec.x` , `vec.y` , `vec.z` e `vec.w` rispettivamente dove ciascuno di essi rappresenta una coordinata nello spazio. Nota che il componente `vec.w` non è usato come posizione nello spazio (abbiamo a che fare con il 3D, non con la 4D) ma è usato per qualcosa chiamato divisione prospettica .



---

Per impostare l'output del Vertex Shader dobbiamo assegnare i dati di posizione alla variabile `gl_Position` predefinita che è di tipo `vec4`.

Alla fine del **main**, qualunque cosa impostiamo su `gl_Position` verrà utilizzata come output del Vertex Shader.

Poiché il nostro input è un vettore di dimensione 3, dobbiamo eseguire il cast di questo in un vettore di dimensione 4.

Possiamo farlo inserendo i valori di `vec3` all'interno del costruttore di `vec4` e impostando il suo componente `w` su `1.0f` (spiegheremo perché successivamente).

Questo è il Vertex Shader più semplice che si possa avere, perché non elabora i dati di input e li inoltra senza modificarli allo step successivo della pipeline.

---



Nelle applicazioni **reali i dati di input non sono generalmente già nelle coordinate del dispositivo normalizzate**, quindi dobbiamo prima trasformare i dati di input in coordinate che rientrano nella regione visibile di OpenGL.

## Compilare uno shader

- Affinché OpenGL utilizzi lo shader , **deve compilarlo dinamicamente in fase di esecuzione** dal suo codice sorgente.
- La prima cosa che dobbiamo fare è creare un oggetto shader, a cui fa riferimento un ID, vertexShader.

```
unsigned int vertexShader;  
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

glCreateShader chiede come argomento il tipo di Shader che si vuole creare . Poiché stiamo creando uno shader di vertici, passiamo GL\_VERTEX\_SHADER .

Quindi collegiamo il codice sorgente dello shader, all'identificativo dell'oggetto shader e compiliamo lo shader:



```
glShaderSource (vertexShader, 1, &vertexShaderSource, NULL);  
glCompileShader (vertexShader);
```

glShaderSource :

**primo argomento:** oggetto shader da compilare.

**secondo argomento:** specifica quante stringhe stiamo passando come codice sorgente, che è solo una.

**terzo parametro:** il codice sorgente effettivo del vertex shader

**quarto parametro :** NULL .



## Fragment Shader

Il Fragment Shader consiste nel calcolare l'output del colore dei pixel. In questo primo esempio, per semplificare le cose, lo shader di frammenti produrrà sempre un colore arancione.

```
#version 420  
  
core  
  
out vec4 FragColor;  
  
void main()  
{  
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);  
}
```

La parola chiave **out** specifica che l'output del fragment color è il vec4 Fragcolor

alfa=1 , completamente opaco



---

Il processo per la compilazione di un fragment shader è simile allo shader di vertice, sebbene questa volta utilizziamo la costante `GL_FRAGMENT_SHADER` come tipo di shader:

- `unsigned int fragmentShader;`
- `fragmentShader = glCreateShader (GL_FRAGMENT_SHADER);`
- `glShaderSource (fragmentShader, 1, &fragmentShaderSource, NULL);`
- `glCompileShader (fragmentShader);`
- Entrambi gli shader sono ora compilati e l'unica cosa che resta da fare è collegare entrambi gli oggetti shader in a programma shader che possiamo usare per il rendering.

## Programma shader

- Un programma shader è la versione finale collegata di più shader combinati. Per usare gli shader appena compilati, dobbiamo collegarli su un programma shader e quindi attivare questo programma shader durante il rendering degli oggetti-



---

La creazione di un oggetto programma è semplice:

```
unsigned int shaderProgram; shaderProgram = glCreateProgram();
```

Il `glCreateProgram` crea un programma e restituisce il riferimento ID all'oggetto programma appena creato.

Ora dobbiamo collegare gli shader precedentemente compilati all'oggetto programma e quindi collegarli `glLinkProgram` :

```
glAttachShader (shaderProgram, vertexShader); glAttachShader  
(shaderProgram, fragmentShader); glLinkProgram (shaderProgram);
```

Il risultato è un programma che possiamo attivare chiamando `glUseProgram` con l'oggetto programma appena creato come argomento:

```
glUseProgram (shaderProgram);
```

Ogni chiamata di disegno dopo `glUseProgram`, utilizzerà questo programma (e quindi gli shader).

---



---

E' possibile eliminare gli oggetti shader dopo averli collegati all'oggetto programma.

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

A questo punto:

i dati del vertice di input sono stati inviati alla GPU;

abbiamo indicato alla GPU come dovrebbe elaborare i dati del vertice all'interno di uno shader di vertici e frammenti.

OpenGL non sa ancora come dovrebbe interpretare i dati dei vertici in memoria e come deve connettere i dati dei vertici agli attributi del Vertex Shader.

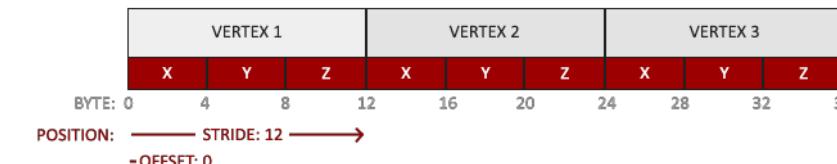
---



## Collegamento degli attributi del vertice

E' necessario specificare manualmente quale parte dei nostri dati di input va a quale attributo di vertice nel Vertex Shader. Ciò significa che dobbiamo specificare come OpenGL dovrebbe interpretare i dati del vertice prima del rendering.

I nostri dati del buffer dei vertici sono formattati come segue:



- I dati di posizione vengono memorizzati come valori in virgola mobile a 32 bit (4 byte).
- Ogni posizione è composta da 3 di questi valori.
- Non c'è spazio (o altri valori) tra ogni set di 3 valori. I valori sono ben confezionati nell'array.
- Il primo valore nei dati è all'inizio del buffer.

Con questa conoscenza possiamo dire a OpenGL come deve interpretare i dati del vertice (per attributo del vertice) usando `glVertexAttribPointer` :

```
glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*) 0);  
 glEnableVertexAttribArray (0);
```



```
glVertexAttribPointer (0, //quale attributo del vertice vogliamo configurare  
                      3, //specifica la dimensione dell'attributo del vertice  
                      GL_FLOAT, //Tipo di dati  
                      GL_FALSE, //specifica se vogliamo i dati normalizzati  
                      3 * sizeof(float), //indica lo stride, cioè l'offset tra due attributi consecutivi del vertice  
                      (void*) 0 //L'ultimo parametro è di tipo void* e quindi è richiesto il cast (void*). Rappresenta l'offset di dove iniziano i dati di posizione nel buffer. Poiché i dati di posizione sono all'inizio dell'array di dati, questo valore è solo 0.  
);  
  
 glEnableVertexAttribArray (0);
```

abilitare l'attributo vertice con `glEnableVertexAttribArray` dandogli la posizione dell'attributo del vertice come argomento.



```
//Vertici del triangolo nel sistema di coordinate normalizzate
```

```
float vertices[] = {  
    // posizioni  
    -0.5f, -0.5f, 0.0f, // vertice in basso a sinistra  
    0.5f, -0.5f, 0.0f, // vertice in basso a destra  
    0.0f, 0.5f, 0.0f // vertice in alto  
};
```



```
void INIT_VAO()
{
    //Genero un VAO
    glGenVertexArrays(1, &VAO);
    //Ne faccio il bind (lo collego, lo attivo)
    glBindVertexArray(VAO);

    //AL suo interno genero un VBO
    glGenBuffers(1, &VBO);
    //Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    //Carico i dati vertices sulla GPU
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Configurazione dell'attributo posizione: informo il vertex shader su: dove
    // trova le informazioni sulle posizioni e come le deve leggere
    //dal buffer caricato sulla GPU
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

}
```



```
void drawScene(void)
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glBindVertexArray(VAO); //Rende attivo il VAO
    glDrawArrays(GL_TRIANGLES, //Assembla a 3 a 3 I vertici in un Triangolo
    0, //a partire dalla posizione 0
    3 //3 vertici );
    glBindVertexArray(0); //Disattiva il VAO
    glutSwapBuffers(); //swap tra il frame buffer front e back durante
    un'animazione
}
```